Optimizing Array Accesses in High Productivity Languages

Mackale Joyner, Zoran Budimlić, and Vivek Sarkar

Rice University, Houston TX. {mjoyner, zoran, vsarkar}@cs.rice.edu

Abstract. One of the outcomes of DARPA's HPCS program has been the creation of three new high productivity languages: Chapel, Fortress, and X10. While these languages have introduced improvements in language expressiveness and programmer productivity, several technical challenges still remain in delivering high performance with these languages. In the absence of optimization, the high-level language constructs that improve productivity can result in order-of-magnitude runtime performance degradations.

This paper addresses the problem of efficient code generation for high level array accesses in the X10 language. Two aspects of high level array accesses in X10 are important for productivity but also pose significant performance challenges: the high level accesses are performed through Point objects rather than integer indices, and variables containing references to arrays are rank-independent. Our solution to the first challenge is to extend the X10 compiler with *automatic inlining and scalar replacement of Point objects*. Our partial solution to the second challenge is to use X10's *dependent type system* to enable the programmer to annotate array variable declarations with additional information for the rank and region of the variable, and to allow the compiler to generate efficient code in cases where the dependent type information is available. Although this paper focuses on high level array accesses in X10, our approach is applicable to similar constructs in other languages.

Our experimental results for single-thread performance demonstrate that these compiler optimizations can enable high-level X10 array accesses with implicit ranks and Points to improve performance by up to a factor of 5.4× over unoptimized X10 code, and to also achieve performance comparable (from 48% to 100%) to that of lower-level Java programs. These results underscore the importance of the optimization techniques presented in this paper for achieving high performance with high productivity.

1 Introduction

The Defense Advanced Research Projects Agency (DARPA) has challenged supercomputer vendors to increase development productivity in high-performance scientific computing by a factor of 10 by the year 2010. DARPA has recognized that constructing new languages designed for scientific computing is important to meeting this productivity goal. Cray (Chapel), IBM (X10), and Sun (Fortress)

have developed new high productivity languages in response to this challenge. While these languages' abstractions suitably provide the mechanisms necessary to improve productivity in high-performance scientific computing [10], compiler optimizations are crucial to minimizing performance penalties resulting from the abstractions.

This paper addresses the problem of efficient code generation for high level array accesses in the X10 language. There are two aspects of high level array accesses in X10 that are important for productivity but that also pose significant performance challenges. First, the high level accesses are performed through Point objects rather than integer indices. Points support an object-oriented approach to specifying sequential and parallel iterations over general array regions and distributions in X10. As a result, the Point object encourages programmers to implement reusable high-level iteration abstractions to efficiently develop array computations for scientific applications without having to manage many of the details typical for low level scientific programming. However, the creation and use of new Point objects in each iteration of a loop can be a significant source of overhead. Second, variables containing references to arrays are rank-independent *i.e.,* by default, the declaration of an array reference variable in X10 does not specify the rank (or dimension sizes) of its underlying array. This makes it possible to write rank-independent code in X10, but poses a challenge for the compiler to generate efficient rank-specific code. Our solution to the first challenge is to extend the X10 compiler so as to perform *automatic inlining and scalar replacement of Point objects*. We have a partial solution to the second challenge that uses X10's *dependent type system* to enable the programmer to annotate selected array variable declarations with additional information for the rank and region of the variable, and to extend the compiler so as to generate efficient code in cases where the dependent type information is available. In the future, we plan to evaluate existing algorithms in the literature for rank and region analysis to test their effectiveness for X10 arrays, so as to reduce the need for the programmer to provide the dependent type annotations.

Our experimental results for single-thread performance demonstrate that these compiler optimizations can enable high-level X10 array accesses with implicit ranks and Points to improve performance by up to a factor of 5.4× over unoptimized X10 code, and to also achieve performance comparable (from 48% to 100%) to that of lower-level Java programs. Even though the current prototype X10 implementation [18] targets Java as its execution platform, we expect the code optimizations presented here to be applicable to other source languages (including Chapel and Fortress) and other target languages (including C and FORTRAN). Further, recent improvements in Java optimization and implementation technologies show that Java performance can also approach that of native FORTRAN and C for some high-performance scientific applications [15]. Thus, we believe that the experimental results in this paper are also indicative of the impact that the optimizations will have on future production-strength implementations of the new high-productivity languages.

Section 2 discusses X10 language constructs related to arrays, points, regions, and point-wise loops. Section 3 describes the optimizations we utilize to enhance the performance of applications employing these specific language constructs. Finally, section 4 presents the experimental results obtained from these compiler optimizations.

2 X10 Language Overview

In this section, we summarize X10 features related to *arrays*, *points*, *regions* and *loops* [7], and discuss how they contribute to improved productivity in high performance computing. Since the introduction of arrays in the FORTRAN language, the prevailing model for arrays in high performance computing has been as a contiguous sequence of elements that are addressable via a Cartesian index space. Further, the actual layout of the array elements in memory is typically dictated by the underlying language *e.g.,* column major for FORTRAN and row major for C. Though this low-level array abstraction has served us well for several decades, it also limits productivity due to the following reasons:

1. *Iteration.* It is the programmer's responsibility to write loops that iterate over the correct index space for the array. Productivity losses can occur when the programmer inadvertently misses some array elements in the iteration or introduces accesses to non-existent array elements (when array indices are out of bounds).
2. *Sparse Array accesses.* Iteration is further complicated when the programmer is working with a logical model of sparse arrays, while the low level abstraction supported in the language is that of dense arrays. Productivity losses can occur when the programmer introduces errors in managing the mapping from sparse to dense indices.
3. *Language Portability.* The fact that the array storage layout depends on the underlying language (*e.g.,* C vs. FORTRAN) introduces losses in productivity when translating algorithms and code from one language to another.
4. *Limitations on Compiler Optimizations.* Finally, while the low-level array abstraction can provide programmers with more control over performance, there is a productivity loss incurred due to its interference with the compiler's ability to perform data transformations for improved performance (such as array dimension padding and automatic selection of hierarchical storage layouts).

The X10 language addresses these productivity limitations by providing higher-level abstractions for arrays and loops that build on the concepts of *points* and *regions* (which were in turn inspired by similar constructs in languages such as ZPL). 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 an iteration construct such as the point-wise for loop. The benefits of using points inside of *for* loops include: potential reuse of common iteration patterns via storage inside

of regions and simple point references replacing multiple loop index variables to access array elements. We use the term, *compact region*, to refer to a region for which the set of points can be specified in bounded space[1], independent of the number of points in the region. Rectangular, triangular, and banded diagonal regions are all examples of compact regions. In contrast, sparse array formats such as compressed row/column storage are examples of non-compact regions.

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 (a one-dimensional region)
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 (will be rectangular if R1 and R2 are rectangular)
R1 || R2 ::= union of regions R1 and R2 (may or may not be rectangular,compact)
R1 - R2 ::= region difference (may or may not be rectangular,compact)
```

Array operations:

```
A.rank    ::= # dimensions in array
A.region ::= index region (domain) of array
A.distribution ::= distribution of array A
A[P] ::= element at point P, where P belongs to A.region
A | R ::= restriction of array onto region R (returns copy of subarray)
A.sum(), A.max() ::= sum/max of elements in array
A1 <op> A2 ::= returns result of applying a point-wise op on array elements,
               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. 1. Region operations in X10

Points and regions are first-class value types [1] in X10 — a programmer can declare variables and create expressions of these types using the operations listed in Figure 1. 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 corre-

[1] For this purpose, we assume that the rank of a region can be assumed to be bounded.

spond 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 1. A new array can be created by restricting an existing array to a sub-distribution, by combining multiple arrays, and by performing point-wise operations on arrays with the same region. Note that the X10 array allocation expression, "`new double[R]`", directly allocates a multi-dimensional array specified by region `R`. In its full generality, an array allocation expression in X10 takes a *distribution* instead of region. However, we will ignore distributions in this paper, since we limit our attention to single-place executions.

As an example, consider the Java and code fragments shown in Figure 2 for the Java Grande Forum [12] SOR benchmark[2]. Note that the Java version involves a lot of manipulation of explicit array indices and loops bounds that can be error prone. In contrast, the rank-specific X10 version uses a single `for` loop to iterate over all the points in the inner region (`R_inner`), and also uses point expressions of the form "`t+[-1,0]`" to access individual array elements. One drawback of the *point-wise* `for` loop in the X10 version is that (by default) it leads to an allocation of a new point object in every iteration for the index and for each subscript expression, thereby significantly degrading performance. Fortunately, the optimization techniques presented in this paper enable the use of point-wise loops as in the bottom of Figure 2, while still delivering the same performance as manually indexed loops as in the top of Figure 2.

Figure 2 also contains a *rank-independent* X10 version. In this case, an additional loop is introduced to compute the weighted sum using all elements in the stencil. Note that the computation performed by the nested `t` and `s for` loops in this version can be reused unchanged for different values of `R_inner` and `stencil`.

3 Improving Performance of Applications with X10 Language Abstractions

This section has two areas of focus. First, we discuss a compiler optimization we employ to reduce the overhead of using *points* in X10. Second, we use X10's *dependent type system* to further improve code generation. As an example, Figure 3 contains a simple code fragment illustrating how X10 arrays may be indexed with points in lieu of loop indices. Figure 4 shows the unoptimized Java output generated by the reference X10 compiler [18] from the input source code in Figure 3. The *get* and *set* operations inside the *for* loops are expensive, and this is further exacerbated by the fact that they occur within innermost loops.

To address this issue, we have a developed an optimization that is a form of *object inlining*, specifically tailored for value-type objects. Object inlining [2,

[2] For convenience, we use the same name, `G`, for the allocated array as well as the array used inside the SOR computation, even though the actual benchmark uses distinct names for both.

Java version:

```
double G[][] = new double[M][N];
. . .
int Mm1 = M-1; int Nm1 = N-1;
for (int p=0; p<num_iterations; p++) {
   for (int i=1; i<Mm1; i++) {
      double[] Gi = G[i]; double[] Gim1 = G[i-1]; double [] Gip1 = G[i+1];
      for (int j=1; j<Nm1; j++)
         Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1] + Gi[j+1])
                 + one_minus_omega * Gi[j];
   } // for i
} // for p
```

X10 version (rank-specific):

```
region R = [0:M-1,0:N-1]; double[.] G = new double[R];
. . .
region R_inner = [1:M-2,1:N-2]; // R_inner is a subregion of R
for (int p=0; p<num_iterations; p++) {
   for (point t : R_inner) {
         G[t] = omega_over_four * (G[t+[-1,0]] + G[t+[1,0]]
                + G[t+[0,-1]] + G[t+[0,1]]) + one_minus_omega * G[t];
   } // for t
} // for p
```

X10 version (rank-independent):

```
. . .
region R_inner = ... ; // Inner region as before
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;
   } // for t
} // for p
```

Fig. 2. Java Grande SOR benchmark

```
region arrayRegion1 = [0:datasizes_nz[size]-1];
...
//X10 for loop
for (point p : arrayRegion1) {
   row[p] = rowt[p];...
   col[p] = colt[p];...
   val[p] = valt[p];...
}
```

Fig. 3. X10 source code of loop example taken from the Java Grande sparsematmult benchmark

```
//X10 for loop body translated to Java
for  ... {
   ... // Includes code to allocate a new point object for p
  (row).set(((rowt).get(p)),p);...
  (col).set(((colt).get(p)),p);...
  (val).set(((valt).get(p)),p);...
}
```

Fig. 4. Java source code of loop following translation from X10 to Java by X10 compiler

4, 8, 9] is a compiler optimization for object-oriented languages that transforms objects into primitive data, and the code that operates on objects into code that operates on inlined data. Budimlić [2] and Dolby [8] introduced object inlining as an optimization for Java and C++. General object inlining requires complex escape analysis and concrete type inference, and the transformation is irreversible (once unboxed, objects in general cannot be "reboxed").

However, because points in X10 are value types, we can safely optimize all array accesses utilizing point objects by replacing them with an object inlined point array access version. A value object has the property that once the program initializes the object, it cannot subsequently modify any of the object's fields. This prevents the possibility of the code modifying *point p* in Figure 3 in between the assignments – a situation that would prevent the inlining of the point. As a result, we can inline the point object declared in the *for* loop header. Figure 5 shows the results of applying this point optimization to the loop we introduce in Figure 3, and Figure 6 shows the resulting Java code.

3.1 Point Inlining Algorithm

We perform a specialized version of object inlining [2] to inline *points*. There are two main differences between points and the objects traditionally considered as candidates for object inlining. First, a point variable can have an arbitrary number of fields because a programmer may use points to access arrays of different rank. Second, a point variable may appear in an X10 loop header. Consequently, the specialized object inlining algorithm must transform the X10 loop header by

```
//X10 optimized for loop
for (int i = 0;  i <= datasizes_nz[size] -1; i +=1) {
  // No point allocation is needed here
  row[i] = rowt[i];...
  col[i] = colt[i];...
  val[i] = valt[i];...
}
```

Fig. 5. X10 source code following optimization of X10 loop body

```
//X10 optimized for loop translated to Java
for (int i = 0;  i <= datasizes_nz[size] -1; i +=1) {
  (row).set(((rowt).get(i)),i);...
  (col).set(((colt).get(i)),i);...
  (val).set(((valt).get(i)),i);...
}
```

Fig. 6. Java source code of loop following translation of optimized X10 to Java by X10 compiler

using the inlined point fields as loop index variables. As a result, this may lead to nested *for* loops if the point variable is a multi-dimensional point.

Figure 7 shows the point inlining algorithm. The first step in the algorithm is to use type analysis to discover the rank of all X10 points in the program. Recall, developers may omit rank information when declaring X10 points. However, we need to infer rank information to inline the point. We obtain rank information for points from both point assignments and array domain information found in X10 loop headers. Because points have the value type property, we inline/unbox every point with an inferred rank. When encountering method calls passed point arguments, we reconstruct the inlined point by creating a new point instance, but ensure that this overhead is only incurred on paths leading to the method calls by allowing the code to work with both original and unboxed versions of the point. Finally, when possible, we convert a point-wise X10 loop into a set of nested *for* loops using the X10 loop's range information for each dimension in the region.

3.2 Use of Dependent Type Information for Improved Code Generation

When examining the Java code generated for the optimization example discussed in the previous section (Figure 6) we see that even though the point object has been inlined, significant overheads still remain due to the calls to the get/set methods. These calls are present because by default, the declaration of an array reference variable in X10 does not specify the rank (or dimension sizes) of its underlying array. This makes it possible to write rank-independent code in X10,

```
//flow-insensitive point inlining algorithm

//init pass
for each region r
  r's rank = TOP
for each point p
  p's rank = TOP

//gather rank information
for each AST node n
  case(assignment)
    if (n.lhs == point OR region)
      n.lhs rank = merge(n.lhs's rank, n.rhs's rank)
  case(x10 loop)
    point p = s.formal();
    region r = s.domain();
    p's rank = merge(p's rank, r's rank);

//merge rank using lattice
merge(rank l, rank r) {
  return l ^ r where :
  TOP ^ r = r;
  BOTTOM ^ r = BOTTOM;
  c1 ^ c2 = c1, if c1 equals c2 else BOTTOM;
}

//inline points
for each AST node n
  if (get_rank(n) == CONSTANT) //inlineable point found
    switch(n)
      case(point declaration)
        inline(n);
      case(point use)
        inline(n);
      case(method call argument)
        reconstruct_point(n);
      case(loop with formal point)
        convert_loop(loop)
```

Fig. 7. Algorithm for X10 point inlining

but poses a challenge for the compiler to generate efficient rank-specific code. In this example, all regions and array accesses are one-dimensional, so it should be possible for the compiler to generate code with direct array accesses instead of method calls. Ideally, this information should be deduced automatically by the compiler (*e.g.,* by propagating rank information from the array's allocation site to all its uses), but in general it requires intra- and inter-procedural rank and region analysis of X10 programs which is beyond the scope of this paper and a subject for future work. Instead, the partial solution in this paper is to use the *dependent type system* [11] available in version 1.01 of the X10 language [16] to enable the programmer to annotate selected array variable declarations with additional information for the rank and region of the variable, and to extend the X10 compiler so as to generate efficient code in cases where the dependent type information is available. A key advantage of dependent types over pragmas is that type soundness is guaranteed statically with dependent types, and dynamic casts can be used to limit the use of dependent types to performance-critical code regions.

```
// X10 array declarations with dependent type information
//   rank==1   ==> array is one-dimensional
//   rect      ==> array's region is dense (rectangular)
//   zeroBased ==> lower bound of array's region is zero
double[: rank==1 && rect && zeroBased ] row = ... ;

. . .

region arrayRegion1 = [0:datasizes_nz[size]-1];
//X10 for loop
for (point p : arrayRegion1) {
   row[p] = rowt[p];...
   col[p] = colt[p];...
   val[p] = valt[p];...
}
```

Fig. 8. X10 for loop example from Figure 3, extended with dependent type declarations

```
//X10 optimized for loop translated to Java
for (int i = 0;  i <= datasizes_nz[size] -1; i +=1) {
  ((DoubleArray_c) row).arr_[i] = ((DoubleArray_c) rowt).arr_[i] ;...
  ((DoubleArray_c) col).arr_[i] = ((DoubleArray_c) colt).arr_[i] ;...
  ((DoubleArray_c) val).arr_[i] = ((DoubleArray_c) valt).arr_[i] ;...
}
```

Fig. 9. X10 for loop body translated from X10 to Java by X10 compiler

To illustrate this approach, Figure 8 contains an extended version of the original X10 code fragment in Figure 3 with a dependent type declaration shown for array `row`. Similar declarations need to be provided for the other arrays as well. The X10 compiler ensures the soundness of this type declaration i.e., it does not permit the assignment of any array reference to `row` that is not guaranteed to satisfy the properties. For the one-dimensional case, we extended the code generation performed by the reference X10 compiler [18] to generate the optimized code shown in Figure 9 for array references with the appropriate dependent type declaration. Performing this optimized code generation for multi-dimensional arrays with dependent types is a subject for future work.

4 Performance Results

We ran all experiments on a 1.25 GHz PowerPC G4 with 1.5 GB of memory using the Sun Java Hotspot VM (build 1.5.0_07-87) for Java 5 with the -Xms2000M -Xmx2000M options to set the heap size to 2 GB (we used < 1.5 GB in practice). We measured performance results on the Java Grande benchmarks. All benchmark results are obtained using the class A versions of the benchmark.

We report results for 3 different versions of the benchmark suite. Version 1 is essentially the original Java version obtained from the Java Grande Forum web site [12] renamed with the *.x10* extension – we use this version as the baseline since the X10 compiler currently translates X10 code into Java. Version 2 is an unoptimized direct translation of the Java version into X10, with all Java arrays converted into X10 arrays and integer subscripts replaced by *points*. Version 3 uses the same input X10 program as in Version 2 but turns on the optimizations described in this paper. All results include runtime array bounds checks, null pointer checks and other checks associated with a Java runtime environment.

Table 1 shows the impact of the optimizations by comparing the performance of Versions 2 and 3. Performance improvements in the range of 1.6× to 5.4× were observed for 7 of 8 benchmarks in Table 1. We observed no improvement in the *series* benchmark because its performance is dominated by scalar (rather than array) operations.

Table 2 compares the performance of the Java baseline (Version 1) with the optimized X10 (Version 3) by reporting the execution time ratio for Version 1 relative to version 3. For 6 of 8 benchmarks the ratio is in the range of 0.48 to 1.00, showing that the performance gap is at most a factor of 2 for these benchmarks. For the two remaining benchmarks, *lufact* and *sor*, the ratio is 0.07 indicating that the Java version is 14.3× faster than the X10 version in these two cases. This gap is primarily due to the multi-dimensional array computations in the two benchmarks, and the fact that the efficient code generation discussed in Section 3.2 currently does not support arrays with rank > 1. Enabling efficient code generation for multi-dimensional X10 arrays and comparison to C/Fortran benchmark versions is a subject for future work.

Table 1. Results from optimizing points in X10 version of Java Grande benchmarks

Benchmarks	*Runtime Performance in seconds*		*Speedup Factor*
	Unopt. X10 (Version 2)	*Opt. X10 (Version 3)*	*(Version 2)/(Version 3)*
sparsematmult	57.97	13.83	4.1×
crypt	8.14	4.79	1.7×
lufact	52.87	18.86	2.8×
sor	508.49	93.41	5.4×
series	19.01	18.95	1.0×
moldyn	2.39	1.19	2.0×
montecarlo	7.59	3.49	2.2×
raytracer	2.27	1.43	1.6×

5 Related Work

Object Inlining [2, 4, 8, 9] is a compiler optimization for object-oriented languages that transforms objects into primitive data, and conversely the rest of the program code that operates on objects into code that operates on inlined data. It

Table 2. Comparison of applied compiler optimizations to X10 array point accesses versus the original version with Java arrays

Benchmarks	*Runtime Performance in seconds*		*Performance Ratio*
	Orig. Java (Version 1)	*Opt. X10 (Version 3)*	*(Version 1)/(Version 3)*
sparsematmult	9.75	13.83	0.71
crypt	4.60	4.79	0.96
lufact	1.38	18.86	0.07
sor	6.06	93.41	0.07
series	19.01	18.95	1.00
moldyn	0.57	1.19	0.48
montecarlo	3.00	3.49	0.86
raytracer	1.28	1.43	0.90

is closely related to "unboxing" [14] for functional languages. Budimlić [2] and Dolby [8] introduced object inlining as an optimization for object-oriented languages, particularly for Java and C++. General object inlining requires complex escape analysis and concrete type inference, and the transformation is irreversible (once unboxed, objects cannot always be reboxed). Joyner [6, 13] extended the analysis to allow more objects and arrays of objects to be inlined in scientific, high performance Java programs. This paper presents object inlining for *points* and other *value* objects in X10, which is a less general, but more effective and more applicable (*all* value objects can be boxed and unboxed freely) form of object inlining.

Wu et al. [17] presented *Semantic Inlining* for *Complex* numbers in Java, an optimization closely related to object inlining. Their optimization incorporates the knowledge about the semantics of a standard library (Complex numbers) into the compiler, and converting all the Complex numbers into data structures containing the real and imaginary part. Although this optimization achieves the same effect as object inlining for Complex numbers, it is less general since it requires compiler modifications for any and all types of objects for which one desires to apply this optimization.

The point-wise *for* loop language abstraction is not unique to the X10 language. Titanium [19], a Java dialect, also has *for* loops which iterate over points in a given domain. The Titanium compiler also performs an optimization to remove points appearing inside *for* loops. However, there are a couple of differences between our approach and the one applied in Titanium. First, because in X10 the rank specification of both points and arrays is not required at the declaration site, we employ a type analysis algorithm to determine the rank for all X10 arrays. Second, object inlining in X10 is directly applicable to all *value* objects, not just points, and thus is a more general optimization.

6 Conclusions and Future Work

In this paper, we discussed the Point abstraction in high-productivity languages, and described compiler optimizations that reduce their performance overhead. We conducted experiments that validate the effectiveness of our optimizations and demonstrate that these optimizations can enable high-level X10 array accesses written with implicit ranks and Points to achieve performance comparable to that of low-level programs written with explicit ranks and integer indices. The experimental results showed performance improvements in the range of 1.6× to 5.4× for 7 of 8 Java Grande benchmark programs written in X10, as a result of these optimizations. Further, for 6 of 8 benchmarks, the performance ratio of the optimized X10 versions relative to the low-level Java versions was in the range of 0.48 to 1.00, showing that the performance gap is at most a factor of 2 for these benchmarks. These results emphasize the importance of the optimizations we have presented in this paper as a step towards achieving high performance for high productivity languages.

We plan to investigate possible optimizations to the X10 array implementation that brings it closer to Java array performance. We will be exploring the ways to communicate static compiler analysis information to the run-time environment to further speed up array accesses, for example by eliminating array bounds checks whenever possible.

We will examine the achievability of an object inlining framework that would expand inlining to more general types of objects. This framework will require a sophisticated concrete type analysis for high-productivity languages, which is an exciting problem in its own right.

Acknowledgments

We would like to thank the anonymous reviewers for their detailed feedback on the paper.

We are grateful to all X10 team members for their contributions to the X10 software used in this paper. We would like to especially acknowledge Vijay Saraswat's work on the design and implementation of dependent types in X10, and Rajkishore Barik's work on optimized code generation for rectangular loops in X10.

We would also like to acknowledge the contributions of the late Ken Kennedy, who for a long time led a multi-institutional effort of bringing high-productivity and high-performance together and who was particularly enthusiastic about this project and participated in its early stages.

Mackale Joyner and Zoran Budimlić are supported in part by an IBM University Relations Faculty Award. While at IBM, Vivek Sarkar's work on X10 was supported in part by the Defense Advanced Research Projects Agency (DARPA) under its Agreement No. HR0011-07-9-0002.

References

1. Bacon, D.F.: Kava: a Java dialect with a uniform object model for lightweight classes. Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande. Palo Alto, California. 68–77
2. Budimlić, Z.: *Compiling Java for High Performance and the Internet*. PhD thesis. Rice University. (2001)
3. Budimlić, Z., Kennedy, K.: JaMake: A Java Compiler Environment. *In 3rd International Conference on Large Scale Scientific Computing*. (2001) 201–209
4. Budimlić, Z., Kennedy, K.: Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463. (1997)
5. Budimlić, Z., Kennedy, K.: Prospects for Scientific Computing in Polymorphic, Object-Oriented Style. *In the Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*. San Antonio, Texas. (1999)
6. Budimlić, Z., Joyner, M., Kennedy, K.: Improving Compilation of Java Scientific Applications . *The International Journal of High Performance Computing Applications*. (2006)
7. Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., Praun, C.v., Saraswat, V., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*. (2005)
8. Dolby, J.: Automatic Inline Allocation of Objects. *In Proceedings of ACM SIGPLAN conference on POPL*. Las Vegas, Nevada. (1997)
9. Dolby, J., Chien, A.: An Automatic Object Inlining Optimization and its Evaluation. *In Proceedings of the 2000 ACM Sigplan Conference on Programming Language Design and Implementation*. (2000) 345–357
10. Ebcioglu, K., Sarkar, V., El-Ghazawi, T., Urbanic, J.: An Experiment in Measuring the Productivity of Three Parallel Programming Languages. *HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2006), held in conjunction with HPCA 2006*. (2006)
11. Harper, R., Mitchell, J.C., Moggi, E.: Higher-order modules and the phase distinction. *In POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, New York. 341–354
12. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande.
13. Joyner, M.: Improving Object Inlining for High Performance Java Scientific Applications. *Master's Thesis*. Rice University. (2005)
14. Leroy, X.: Unboxed objects and polymorphic typing. *In Proceedings of the 19th Symposium on the Principles of Programming Languages*. (1992) 177–188
15. Markidis, S., Lapenta, G., VanderHeyden, W.B., Budimlić, Z.: Implementation and Performance of a Particle-in-cell code Written in Java. *Concurrency and Computation: Practice and Experience*, Vol. 17. (2005) 821–837
16. Saraswat, V.: Report on the experimental language x10 version 1.01. http://x10.sourceforge.net/docs/x10-101.pdf
17. Wu, P., Midkif, S., Moreira, J., Gupta. M.: Efficient support for complex numbers in Java. *Proceedings of the ACM 1999 conference on Java Grande*. (1999) 109–118
18. X10 Prototype Implementation. http://x10.sf.net.
19. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, Vol. 10, Issue 11-13. (1998) 825–836