Array Optimizations for Parallel Implementations of High Productivity Languages

Mackale Joyner, Zoran Budimlić, Vivek Sarkar, Rui Zhang

Department of Computer Science, Rice University

{mjoyner, zoran, vsarkar, ruizhang}@cs.rice.edu

Abstract

DARPA's HPCS program has set a goal of bringing high productivity to high-performance computing. This has resulted in the creation of three new high-level languages, namely Chapel, Fortress and X10, that have successfully addressed one aspect of productivity: programmability. Unfortunately, the current state of the art in implementation of these high-level language concepts result in significant performance overheads. Our research addresses this issue by concentrating on the second aspect of productivity: performance.

This paper presents an interprocedural rank analysis algorithm that is capable of automatically inferring ranks of the arrays in X10, a language that allows rank-independent specification of loop and array computations using *regions* and *points*. Further, it uses the rank analysis information to enable storage transformations on arrays; the storage transformation evaluated in this paper converts high-level multidimensional X10 arrays into lower-level multidimensional Java arrays, when legal to do so. We also describe a compiler-to-runtime communication strategy that determines when array bounds checks can be eliminated in high-level X10 loops, and conveys that information to the run-time system, further improving performance. We use a 64-way AIX Power5+ SMP machine to evaluate our optimizations on a set of parallel computational benchmarks and show that they optimize X10 programs with high-level loops using *regions, points* and *rank-free computation* to deliver performance that rivals the performance of lower-level, hand-tuned code with explicit loops and array accesses, and up to two orders of magnitude faster than unoptimized, high-level X10 programs. The experimental results also show that our optimizations help the scalability of X10 programs as well, demonstrating that relative performance improvements over the unoptimized versions increase as we scale from using a single CPU to using all 64 CPUs.

General Terms Compilers

Keywords high productivity, high performance, type analysis

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. Several new language are a result of this initiative: Chapel (Cray), X10 (IBM), and Fortress (Sun).

These languages have significantly improved the programmability of high-performance, scientific codes through a use of higher-level language constructs, object-oriented design, and higher-level abstractions for arrays, loops and distributions [10]. Unfortunately, high programmability often comes at a price of lower performance. These higher-level abstractions and constructs can sometimes result in up to two orders of magnitute longer execution times.

This paper addresses efficient array implementation for high productivity languages, particularly X10. Figure 1 shows the X10 compiler structure assumed in our research. The focus of this paper is on compiler analyses and optimizations that improve the performance of high level array operations in high productivity languages — compilers for other high productivity languages have a similar structure to Figure 1. In Section 3.1 we present an interprocedural array rank analysis algorithm, that automatically infers exact ranks of rank-free array variables in many X10 programs. We also describe an array transformation strategy (Section 3.4), that uses the results from our rank analysis algorithm to convert general X10 arrays into a lower-level, more efficient Java arrays. These two techniques, combined with *object inlining* of *points* [5, 12, 13] result in performance improvements of up to two orders of magnitude.

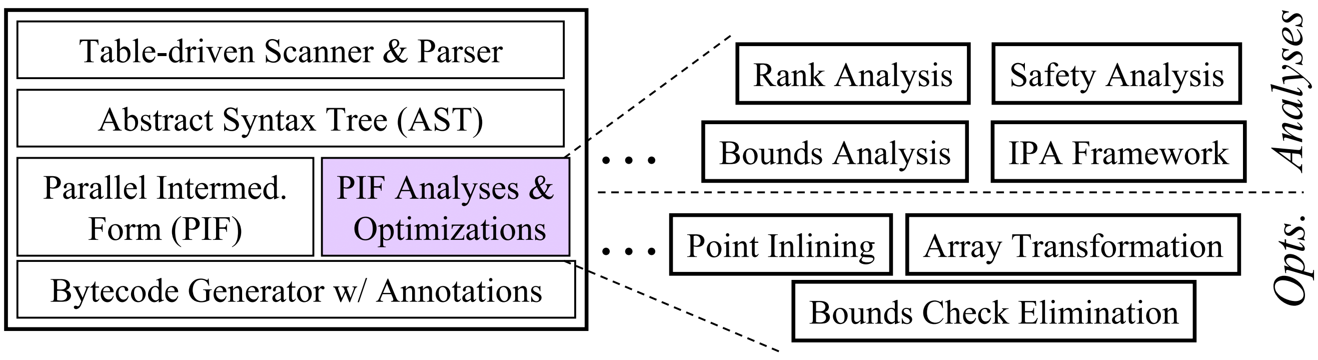
Table-driven Scanner & Parser
Abstract Syntax Tree (AST)
Parallel Intermed. Form (PIF)
PIF Analyses & Optimizations
Bytecode Generator w/ Annotations
Rank Analysis
Safety Analysis
Bounds Analysis
IPA Framework
Analyses
Point Inlining
Array Transformation
Bounds Check Elimination
Opts.

Figure 1. X10 compiler structure

In Section 4, we validate our techniques on a set of parallel Java Grande benchmarks [11]. Several of these benchmarks are rewritten in X10 to use the X10 high-level loop constructs, *points* and *regions* for manipulating arrays, and rank-free array computation in places where such abstractions improve the readability, expressiveness and generality of the program. We show that the combination of our techniques results in very large performance improvements, up to more than two orders of magnitude in some cases. We ran our experiments on a 64-way SMP machine and demonstrated that our techniques not only apply to parallel as well as sequential programs, but that in fact our transformations help improve the scalability of X10 programs by eliminating some synchronization bottlenecks that can occur in current X10 implementation.

To further improve array performance, we have also devised a compiler-to-runtime communication strategy for eliminating array bounds checks, which is discussed in detail in Section 5. This strategy uses the information our compiler infers from high-level X10 loops that use *points* and *regions* to determine which array accesses within the loop do not have to be checked for out of bounds exceptions. We then communicate this information to the X10 run-time system (the JIT compiler) to avoid unecessary array bounds checks. This results in further performance improvements of up to 10%.

Section 7 concludes the results of this paper and suggest some directions for future research.

2. X10 Arrays

X10 provides powerful high-level array syntax that encourages programmers to code array computations in a highly productive manner. As an example, consider the Java code fragment shown in Figure 2 for the Java Grande Forum [11] SOR benchmark[1]. Note that the Java version involves a lot of explicit array index manipulation and explicit 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 in a stencil computation. One consequence 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 all subscript expressions, thereby significantly degrading performance. We previously addressed this problem (for sequential execution only) with a point inlining optimization [13]. However, after applying this transformation, we still experience up to 2 orders of magnitude in performance degradation when comparing Java Grande benchmarks with X10's general high-level arrays against the same benchmarks with lower-level Java arrays.

In addition to point expressions, X10 also promotes productivity by enabling programmers to develop rank-independent array computations. Figure 2 also contains a *rank-independent* X10 version of the SOR array computation. 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`. During compilation, the X10 compiler translates X10 code into Java. The X10 compiler transforms all X10 arrays into objects, and array accesses (read or write) into *get* and *set* method calls. These *get* and *set* method calls can be expensive, especially if they occur within the innermost loop, which is very often the case. The performance penalty can reach a staggering 2 orders of magnitude over lower-level Java version of the code in some cases.

This paper presents a technique for enabling high-level array utilization without suffering reductions in performance. We present an algorithm to automatically determine when the compiler can convert high-level X10 arrays into a more efficient lower-level representation.

[1] 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
```

Figure 2. Java Grande SOR benchmark

3. X10 General Array Conversion

The algorithm for converting general X10 arrays into a lower-level, more efficient representation, consist of three phases. The first phase is Rank Analysis, which infers the concrete ranks of all the X10 arrays in the program. The second phase is Safety Analysis, which determines which X10 arrays can be safely converted into Java arrays, using the rank information computed in Phase 1. The last phase of the algorithm is the actual conversion of the code that manipulates X10 arrays into code that operates directly on the underlying Java arrays. This section describes the three phases of the algorithm in more detail.

3.1 Rank Analysis

This section details the type inference algorithm we utilize to discover the ranks of X10 arrays. Recall, the generality of X10 arrays enables programmers to develop rank independent code by omitting array dimensionality at the declaration site. We need precise rank information to generate more efficient array representations. We employ an interprocedural, context-insensitive algorithm to gather array rank information from the X10 program.

Our whole-program analysis algorithm uses intraprocedural analysis to capture local rank information from examining array assignments. We then perform interprocedural analysis to glean rank information arising from both X10 array method arguments and methods returning X10 arrays.

Figure 3 shows the high-level description of the rank inference algorithm.

```
Input: X10 program
Output: $rank$, a mapping of each X10 array, region and point
        to its rank
begin
    // initialization
    foreach Region $r$ do $rank[r] = TOP$
    foreach Point $p$ do $rank[p] = TOP$
    foreach Array $a$ do $rank[a] = TOP$
    // infer X10 array ranks
    foreach AST node $n$ do switch typeof($n$) do
        case assignment
            // get inferred rank for rhs
            $rank \leftarrow TOP$
            if n.lhs is an Array then
                switch typeof($n.rhs$) do
                    case array constructor
                        $rank \leftarrow$
                        $rank[constructorRegion(n.rhs)]$
                    case method call
                        $rank \leftarrow rank[n.rhs.returnType]$
                    case variable
                        $rank \leftarrow rank[n.rhs]$
                    case array access
                        $rank \leftarrow rank[n.rhs.array] - 1$
            if n.lhs is a Point or Region then
                $rank \leftarrow rank[n.rhs.constructor]$
            $rank \leftarrow merge(rank[n.lhs], rank)$
            $rank[n.lhs] \leftarrow rank$
        case procedure call
            // merge actual with formal parameters
            foreach formal parameter fp and corresponding
            argument ap do
                $rank[fp] \leftarrow merge(rank[ap], rank[fp])$
        case array access
            $rank[n.array] \leftarrow$
            $merge(rank[n.array], rank[n.point])$
        case method return
            $m \leftarrow$ method we're currently analysing
            $rank[m.returnType] \leftarrow$
            $merge(rank[m.returnType], rank[n.expr])$
end
```

Figure 3. Inferring ranks of general X10 arrays

```
Function merge(Rank left, Rank right)
    if left = right or right = $\top$ then return left;
    if left = $\top$ then return right;
    else return $\bot$;
```

The rank information flows from right to left in the rank inference algorithm. That is to say, in an assignment, the inferred rank of the left hand side is the lower (in the type lattice sense) of the rank of the right hand side and the previous rank of the left hand side. Similarly for a method call (in which the parameter passing can be conceptually thought of as assignments of actual parameters to formal parameters), the rank information flows from actual to formal parameters.

The rank inference algorithm can be implemented to run in $O(n)$ time, where n is the number of AST nodes across all procedures in the program. A simple worklist algorithm puts an AST node on the worklist every time an element of the expression has its rank lowered. Since the rank lattice is of finite height (3), the rank of each AST node con be lowered at most 2 times ($\top$ to a number to $\bot$), each AST node will go on the worklist at most 3 times, resulting in $O(n)$ algorithm complexity.

3.2 Safety Analysis

In addition to gathering precise rank information, our type inference algorithm also employs a safety analysis algorithm to ensure that it is safe to transform an X10 general array into a more efficient representation. The alternate representation we currently use is the Java array. There are several X10 array operations that are non-trivial to translate to Java array operations. In these cases, we mark the X10 array as *unsafe*. This ensures that our transformation preserves the original representation of the X10 array. Figure 5 shows the high-level description of the safety analysis algorithm we perform before transforming X10 arrays to Java arrays.

Another detail worth mentioning is that our algorithm performs a two-way safety inference. That is, we utilize safety information on the left hand side of an assignment to infer safety information for the right hand side and vice versa, thereby reducing safety analysis to an equivalence partitioning problem. Our algorithm incorporates this two-way strategy for method arguments and formal parameters as well.

3.3 Extensions for Increased Precision

The Rank analysis and Safety analysis algorithms as presented in this section are fairly easy to understand and easy to implement as linear-time flow-insensitive and context-insensitive algorithms. We have also designed more complex flow-sensitive and context-sensitive versions of these algorithms that can potentially compute more precise rank and safety information, and lead to better optimization.

For the set of applications we used as benchmarks in this paper these extensions do not produce more precise results, thus we chose to omit a more detailed discussion of those extensions. Here, we will give a brief description of those techniques.

SSA Form The Rank Analysis and Safety Analysis algorithms as described on Figures 3 and 5 are flow insensitive. Thus, if an array variable a is reassigned an array of a different rank than before, it will get $\bot$ as its rank, which can futher get propagated to other variables involved in computation with a. Similarly, if a variable is marked unsafe for conversion into a Java array, it will prevent conversion of all occurences of that variable into a Java array, even if they could potentially be safely converted in different regions of the code. This source of imprecision can be eliminated by converting the code into SSA form [4]. The ϕ nodes in the SSA form are treated similarly to an assignment: the rank of the variable on the left hand side gets assigned a $merge()$ of the ranks of all the argument variables to the ϕ function. Since rank analysis is just an analysis and does not involve any code reorganization, conversion from the SSA

```
Input: X10 program
Output: function $Safe(a)$, $\top$ if X10 array $a$ can be safely
        converted to a Java array
begin
    foreach AST node $n$ do $Safe(n) \leftarrow \top$
    foreach AST node $n$ do switch typeof($n$) do
        case assignment
            if $rank(n.lhs) \neq rank(n.rhs)$ then
                $Safe(n.lhs) \leftarrow Safe(n.rhs) \leftarrow \bot$
            if $\neg Safe(n.lhs)$ then $Safe(n.rhs) \leftarrow \bot$
            else switch typeof($n.lhs$) do
                // region must be rectangular, zero
                   based
                case region
                    $Safe(n.lhs) \leftarrow Rectangular(n.rhs) \wedge$
                    $ZeroBased(n.rhs)$
                case array
                    if $n.rhs$ is a constructor then
                        $Safe(n.lhs) \leftarrow$
                        $Safe(getRegion(n.rhs))$
                    if $n.rhs$ is a variable then
                        $Safe(n.lhs) \leftarrow Safe(n.rhs)$
        case procedure call
            // only allow region rank() calls
            if $n.target$ is region then
                $Safe(n.target) \leftarrow isRankCall(n)$
            // only allow array region() calls
            if $n.target$ is array then
                $Safe(n.target) \leftarrow isRegionCall(n)$
            // check safety of array arguments
            foreach formal parameter $fp$ and corresponding
            argument ap do
                $safe \leftarrow rank(fp) = rank(ap)$
                $safe \leftarrow safe \wedge \neg$ ($fp$ or $ap$ is a region)
                if $fp$ or $ap$ is an array then
                    $safe \leftarrow safe \wedge Safe(fp) \wedge Safe(ap)$
                $Safe(fp) \leftarrow Safe(ap) \leftarrow safe$
        case binary operation
            // cannot transform X10 arrays involved
               in binary array operations
            $safe \leftarrow \neg$ ($n.lhs$ or $n.rhs$ is array or region)
            $Safe(n.lhs) \leftarrow Safe(n.rhs) \leftarrow safe$
end
```

Figure 5. Safety analysis for transforming X10 arrays into Java arrays

form back into the original form is simple and doesn't involve any copy coalescing [6].

Type Jump Functions The two analysis algorithms as described in this section will propagate the rank and safety information through infeasible paths through the call graph. If a method is called in one place with an argument of rank 2, and on another place with an argument of rank 1, the formal argument will get $\perp$ for the rank, and possibly propagate this lower type through the return variable back into the caller code.

This imprecision can be avoided by using *type jump functions* [8] for method calls. The idea behind type jump functions is to encapsulate the relation between the types of actual arguments to a method and the type of the return argument. Since rank and safety information are essentially types, this method generalization can be used to increase the precision of the rank and safety analysis algorithms. If a type jump function describes a method m as accepting the argument of rank R and returning a value of rank $R-1$, then this method can be analyzed independently at different call sites and will propagate the correct values for the rank, even if the ranks of the arguments at different call sites are different.

During the conversion of X10 arrays into Java arrays, a method with polymorphic rank arguments has to be cloned to a variant with the specific ranks that are determined by the call site. The most aggressive approach is to convert as many X10 arrays as possible by generating as many variants of the method as there are call sites with different sets of ranks for actual arguments. Alternatively, to avoid code explosion, the compiler can generate a limited set of variants for the most profitable call paths, and leave the default variant that uses unconverted X10 arrays for the general case.

Type jump functions for the safety analysis, while similar to those for rank analysis, are simpler since the only two "types" a variable can have are *safe* and *unsafe*.

3.4 Array Transformation

Once we have completed the array rank and safety analysis, we begin the transformation from X10 arrays to the more efficient representation (Java array). There are two main steps in this process. First, we convert each declared X10 array to our analyzed precise type. Second, we must convert the *X10ArrayAccess* AST node into the *ArrayAccess* AST node. The X10 compiler makes the distinction between the 2 nodes so that only the *X10ArrayAccess* can accept a *point* expression as an argument. As a result, during the conversion process, we must also convert any *point* valued subscript expression into equivalent integer-valued expressions since we cannot perform a Java array access with a *point* object.

4. Performance Results

The performance results reported in this section were obtained using the following system settings:

- The target system is a IBM 64-way 2.3 GHz Power5+ SMP with 512 GB main memory.
- The Java runtime environment used for all X10 runs is the IBM J9 virtual machine (build 2.4, J2RE 1.6.0) which includes the IBM TestaRossa (TR) Just-in-Time (JIT) compiler [17]. The following internal TR JIT options were used for all X10 runs:
  - Options to enable classes to be preloaded, and each method to be JIT-compiled at a high ("very hot") optimization level on its first execution rather than being first executed by the interpreter.
  - An option to ignore strict confirmance with IEEE floating point.
  - An option to recognize special annotations communicated by the X10 compiler.
- In addition, a special *skip checks* option was used for some of the results to measure the opportunities for optimization. This option directs the JIT compiler to disable all runtime checks (array bounds, null pointer, divide by zero).
- Version 1.5 of the X10 compiler and runtime [21] were used for all executions. This version supports *implicit syntax* [20] for place-remote accesses. In addition, all runs were performed with the number of places set to 1, so all runtime "bad place" checks [7] were disabled.
- The default heap size used was 2GB, with the exception of one set of results that studied the impact of reducing the heap size.
- For all runs, the main program was extended with a three-iteration loop within the same Java process, and the best of the three times was reported in each case. This configuration was deliberately chosen to reduce/eliminate the impact of JIT compilation time, garbage collection and other sources of perturbation in the performance comparisons.

The benchmarks studied in this paper are X10 ports of benchmarks from the Java Grande [11] suite. The benchmarks were executed using the class C input size (largest size). We compare three versions of each benchmark:

1. The *light* version uses X10 concurrency constructs like `async` and `finish`, but directly uses low-level Java arrays as in [2]. It does not support the productivity benefits of higher-level X10 arrays, but instead serves as a performance target for the optimizations presented in this paper.
2. The *unoptimized* version represents direct execution of the unoptimized X10 programs with high-level array constructs, obtained using the X10 reference implementation on SourceForge [21].

3. The *optimized* version uses the same input program as the *unoptimized* case but also includes the optimizations introduced in this paper.

We compare the performance of the optimized X10 general arrays (Version 3) to both the unoptimized version (Version 2) and our baseline (Version 1). Table 1 shows the raw execution times for the unoptimized and optimized versions, and Figure 6 shows the relative speedup obtained due to optimization. As can be seen in Table 1and Figure 6, the performance improvements due to optimization can be very significant, reaching as a high as a factor of 266.

Figure 7 shows the performance of the optimized X10 implementation (Version 3) relative to the light version (Version 1). We see that the performance is at most 10% in one case (SOR) and is less than 2% in all other cases. These results suggest that the optimization techniques presented in this paper can enable programmers to write high-productivity array computations using X10 arrays without suffering substantial performance penalties.

Although our transformations are designed with sequential performance in mind, they also impact parallel performance. Table 2 shows the relative scalability of the Optimized and Unoptimized X10 versions. Since the biggest difference was observed for the *sparsematmult* benchmark, we use Figures 8 and Figure 9 to further study this behavior for that benchmark. Figure 8 illustrates that the optimized *sparsematmult* benchmark scales better than the unoptimized version with an initial mininum heap size of 2 GB. Figure 9 shows that decreasing the initial mininum heap size to the default size (4 MB) further increases the gap in scalability, thereby suggesting that garbage collection is a major scalability limitation for the Unoptimized case. This is further supported by the fact that Unoptimized version allocates a large number of *point* objects that have short life times. The Optimized version mitigates this problem by inlining *point* objects. Note that in all these results, the Optimized speedup is relative to the 1-CPU optimized performance, and the Unoptimized speedup is relative to the 1-CPU unoptimized performance.

Recall, that the X10 compiler translates X10 arrays into Java objects during code generation which in turn leads to heap allocation for both the arrays and points. When most of the heap is allocated, the garbage collector must run to free up space. Because the garbage collector runs sequentially, the entire cost of executing the garbage collector is distributed equally across each CPU. As a result, as we scale from 1 CPU to all 64 CPUs the slope for the speedup relative 1 CPU decreases.

Simply increasing the heap size is not an effective solution as there will typically be some application that will force the garbage collector to run at whatever the heap size is. We observe this behavior even when increasing the minimum heap size for the *sparsematmult* benchmark result for Figure 8 to 2 GB.

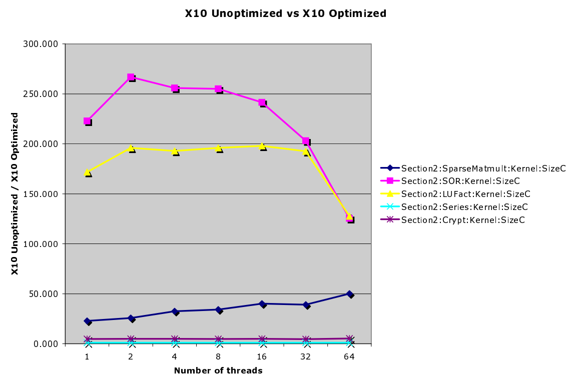

Figure 6. Speedup of Optimized X10 version relative to Unoptimized X10 version

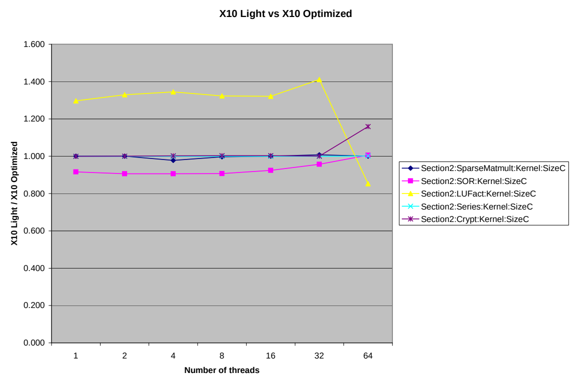

Figure 7. Comparison of the optimized X10 benchmarks relative to the X10 light baseline

5. Automatic Array Bounds Check Elimination

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 array access or buffer overflow. While these checks are beneficial for safety and correctness, performing these checks at run time can significantly degrade performance especially in array-intensive codes.

Significant effort has been made by the compiler research community to statically eliminate array bounds checks in higher-level languages when the compiler can prove that these checks are unnecessary [3, 16, 18]. In this paper, we take advantage of the X10 *region* language abstraction

Table 1. Raw runtime performance of Unoptimized and Optimized X10 versions as we scale from 1 to 64 CPUs

Benchmarks	*Runtime Performance (scaling from 1 to 64 CPUs) in seconds*						
	1	*2*	*4*	*8*	*16*	*32*	*64*
sparsematmult unoptimized	309.495	138.707	73.984	34.269	18.322	10.100	6.306
sparsematmult optimized	13.536	5.398	2.280	1.001	0.457	0.258	0.126
crypt unoptimized	36.388	28.192	13.858	6.902	3.490	1.668	0.971
crypt optimized	7.716	5.732	2.865	1.433	0.717	0.360	0.183
lufact unoptimized	937.158	496.626	238.518	126.333	68.516	42.153	37.257
lufact optimized	5.456	2.535	1.236	0.645	0.346	0.219	0.292
sor unoptimized	614.269	332.917	157.811	81.875	44.658	28.031	23.830
sor optimized	2.757	1.248	0.617	0.321	0.185	0.138	0.190
series unoptimized	1766.129	1767.053	850.969	429.001	215.217	108.858	54.417
series optimized	1764.875	1764.59	850.708	428.981	215.097	107.682	53.786

Table 2. Relative Scalability of Optimized and Unoptimized X10 versions with heap size of 2 GB. The Optimized speedup is relative to the 1-CPU optimized performance, and the Unoptimized speedup is relative to the 1-CPU unoptimized performance.

Benchmarks	*Runtime Performance speedup (relative to 1 CPUs)*						
	1	*2*	*4*	*8*	*16*	*32*	*64*
sparsematmult unoptimized	1.00	2.23	4.18	9.03	16.89	30.64	49.08
sparsematmult optimized	1.00	2.51	5.94	13.52	29.62	52.27	107.43
crypt unoptimized	1.00	1.29	2.63	5.27	10.43	21.82	37.47
crypt optimized	1.00	1.35	2.69	5.38	10.76	21.43	42.16
lufact unoptimized	1.00	1.89	3.93	7.42	13.68	22.23	25.15
lufact optimized	1.00	2.15	4.41	8.46	15.77	24.91	18.68
sor unoptimized	1.00	1.85	3.89	7.50	13.75	21.91	25.78
sor optimized	1.00	2.21	4.47	8.59	14.90	19.98	14.51
series unoptimized	1.00	1.00	2.07	4.12	8.21	16.22	32.46
series optimized	1.00	1.00	2.07	4.11	8.21	16.39	32.81

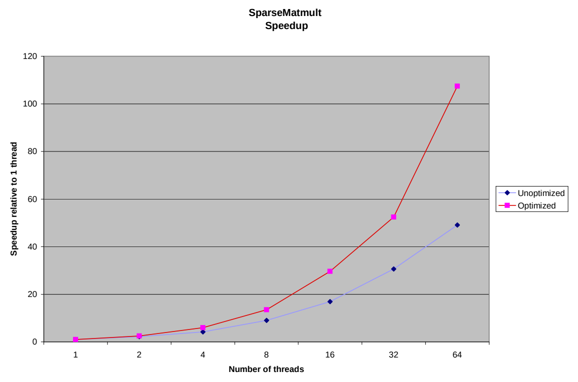

Figure 8. Relative Scalability of Optimized and Unoptimized X10 versions of the sparsematmult benchmark with initial mininum heap size of 2 GB (and maximum heap size of 2GB). The Optimized speedup is relative to the 1-CPU optimized performance, and the Unoptimized speedup is relative to the 1-CPU unoptimized performance.

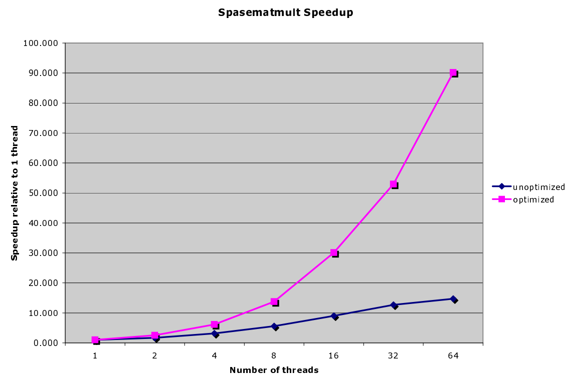

Figure 9. Scalability of Optimized and Unoptimized X10 versions of the sparsematmult benchmark with initial minimum heap size of of 4 MB (and maximum heap size of 2GB). The Optimized speedup is relative to the 1-CPU optimized performance, and the Unoptimized speedup is relative to the 1-CPU unoptimized performance.

```
//sparsematmult loop, no bounds check annotations
//may be added to array accesses inside loop
region arrayRegion1 = [0:datasizes_nz[size]-1];
...
Random R = ...
val = new double[arrayRegion1...]; ...
col = new int[arrayRegion1...]; ...
row = new int[arrayRegion1...]; ...

for (point p : arrayRegion1) {
        // generate random row index (0, M-1)
        row[p] = Math.abs(R.nextInt()) ...

        // generate random column index (0, N-1)
        col[p] = Math.abs(R.nextInt()) ...

        val[p] = R.nextDouble(); ...
}
```

Figure 10. Java Grande SparseMatmult benchmark

to statically determine when array bounds checks are not needed. In these cases, we annotate the array access with a *noBoundsCheck* annotation to signal to the IBM Java J9 VM that it can skip the array bounds check for a particular array access.

X10 regions are particularly beneficial for our static analysis since this language abstraction makes it easier to determine when we may remove an array bounds check. Because regions have the value type property (once defined, cannot subsequently be modified), it is easier to prove that the region the loop iterates over remains unchanged over the entire loop iteration space. For example, consider the two loops:

```
double[.] a = new double[[low,high,str]]
loop1: for (n=low, n < high, str) {
... a[n] ...
}

region r = [low,high,str]
loop2: for (point p : r) {
... a[p] ...
}
```

In *loop1*, we must prove that neither *low*, *high*, nor *str* are changed inside the loop body in a manner that might introduce an illegal array access. However, in *loop2*, this additional analysis is unnecessary since the programmer cannot modify the region bounds or stride. In Figure 10, we illustrate this with a loop taken from the *sparsematmult* benchmark. This is an example where we can apply a transformation that adds annotations to the array accesses, thereby signaling to the VM to skip the bounds check. In order to insert the annotations inside the loop, we must prove two things. First, the domain of *arrayRegion1* in the loop header must contain the domains of all arrays accessed inside the loop with index point *p*. Second, the programmer must not modify the domains of any of these arrays he or she accesses inside the loop with point *p*. In Section 4, we show the results of applying this transformation to a set of benchmarks.

We provide results to first understand the impact runtime checks have on performance and then to demonstrate the benefit static array bounds analysis coupled with *noBoundsCheck* annotations have on runtime execution. We turn off the preload classes flag for these results due to an unresolved issue arising when adding the VM classes needed to recognize our annotations to the classpath. We compare two versions of each benchmark. The skip runtime checks version skips all runtime checks (array bounds, null, divide-by-0). The second version enables these runtime checks. All times reported represent a best of 3 run. Figure 11 indicates that runtime checks (array bounds, null, divide by 0 checks) can degrade performance and only reach 56% of the peak performance when runtime checks are disabled. Therefore, our compiler array bounds analysis has an opportunity to significantly impact runtime performance.

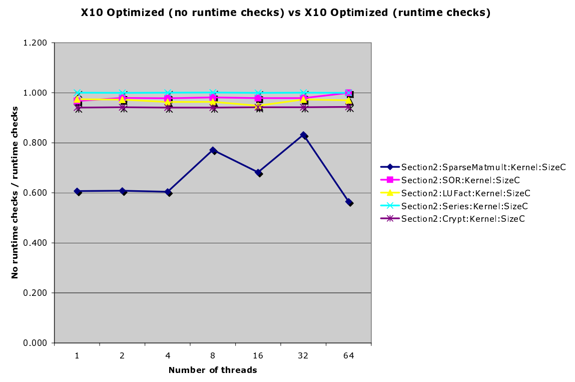

Figure 11. Shows the impact enabling runtime checks has on performance

We next provide preliminary results in Table 3, in the sequential case, to demonstrate the performance impact of automatically adding our *noBoundsCheck* annotations, through static analysis, to our benchmark set with runtime checks enabled. These annotations alert the IBM J9 VM when array bounds checking for an array access is unnecessary. We don't include the *series* benchmark because there is no opportunity to improve the performance with *noBoundsCheck* annotations.

The largest potential impact our no bounds check analysis may have is on the *sparsematmult* benchmark. The other benchmarks still achieve over 90% of runtime performance with the skip runtime checks flag disabled. With our static analysis and subsequent automatic program transformation inserting no bounds check annotations, we increase the runtime performance up to 10%. While this result is encouraging, there are still opportunities to further improve the performance of *sparsematmult* based on results from Figure 11. Figure 12 illustrates the main computation in the *sparsematmult* kernel. In order to annotate each of the arrays inside the loop with a *noBoundsCheck* annotation, we need to prove that actual array values of one array are within the array bounds of another. The addition of this analysis is a subject

Table 3. Results of automatically inserting noBoundsCheck annotations in X10 version of Java Grande benchmarks

Benchmarks	*Sequential Runtime Performance in seconds*				
	Enable Runtime Checks	*Skip Runtime Checks*	*Speedup*	*No Bounds Checks*	Speedup
sparsematmult	22.105	13.986	1.581	20.13	1.098
crypt	8.337	7.781	1.071	8.339	1.000
lufact	5.605	5.055	1.109	5.497	1.020
sor	2.418	2.436	0.993	2.217	1.091

```
//main loop of execution in sparsematmult benchmark
//array indices for all arrays accesses inside loop are
//based on actual array values

region R2 = [lowsum[id]:highsum[id]-1];
 for (int reps=0; reps<NUM_ITERATIONS; reps++) {
     for (point p_run[i] : R2) {
         yt[ row[p_run] ] += x[ col[p_run] ] * val[p_run];
     }
 }
```

Figure 12. Main loop of execution in Java Grande Sparse-Matmult benchmark

for future work. Despite this challenge, our results demonstrate that our static no bounds check analysis helps reduce the performance impact of programmers developing applications in type-safe languages.

6. Related Work

Our type analysis work which generates precise ranks for efficient array representations using equivalence sets is a similar problem to detecting variable or pointer equivalence [1, 14] and copy propagation [19]. Similarly to copy propagation, we have a finite lattice where a variable's lattice value may only drop to a value further down the lattice. The idea of creating specialized method variants based on the calling context is related to specialized library variant generation derived from type jump functions [8]. Our actual transformation from X10 general array to a more efficient form is similar to *object inlining* [5, 9]. We build upon previous work for array bounds checking [3, 15, 16, 18] by using the X10 *region* language abstraction to both reduce the set of variables we need to analyze and to simplify the analysis in determining when array bounds checks are unnecessary.

7. Conclusions and Future Work

This paper makes 3 primary contributions. First, it provides an algorithm to generate rank-specific efficient array computations from applications utilizing productive rank-independent general X10 arrays. The algorithm propagates X10 array rank information to generate the more efficient Java arrays with precise ranks. Our results demonstrate that we can generate efficient array representations and come within 90% of the baseline for each benchmark and within 98% in most cases. In the future, we would like to use equivalent codes written in C and FORTRAN as our baseline. The second contribution shows how our optimizations impact scalability. We illustrate that the optimized version of the benchmarks scale much better than the unoptimized general X10 array version and discuss why this occurs. The final contribution provides both a static array bounds analysis utilizing the X10 *region* and a subsequent program transformation inserting a *noBoundsCheck* annotation when we can prove that performing a Java VM runtime check is unnecessary. These contributions enable programmers to develop high productivity array computations without incurring additional runtime costs associated with utilizing higher level language abstractions.

Acknowledgments

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 Chris Donawa and Allan Kielstra for their help in understanding the IBM J9 VM.

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] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] R. Barik, V. Cave, C. Donawa, A. Kielstra, I. Peshansky, and V. Sarkar. Experiences with an smp implementation for x10 based on the java concurrency utilities (extended abstract). In *Proceedings of the 2006 Workshop on Programming Models for Ubiquitous Parallelism (PMUP), co-located with PACT 2006*, September 2006. www.cs.rice.edu/ vsarkar/PDF/pmup06.pdf.

[3] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[4] P. Briggs, K. Cooper, T. Harvey, and T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881., July 1998.

[5] Z. Budimlić. *Compiling Java for High Performance and the Internet*. PhD thesis, Rice University, 2001.

[6] Z. Budimlić, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. Reeves. Fast copy coalescing and live-range identification. *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 25–32, 2002.

[7] 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.

[8] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. Automatic type-driven library generation for telescoping languages. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[9] J. Dolby. Automatic inline allocation of objects. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 7–17, New York, NY, USA, 1997. ACM Press.

[10] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, and J. Urbanic. An experiment in measuring the productivity of three parallel programming languages. In *HPCA Workshop on Productivity and Performance in High-End Computing, held in conjunction with HPCA*, 2006.

[11] The Java Grande forum benchmark suite. . http://www.epcc.ed.ac.uk/javagrande.

[12] M. Joyner. Improving object inlining for high performance java scientific applications. Master's thesis, Rice University, 2005.

[13] 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.

[14] D. Liang and M. J. Harrold. Equivalence analysis and its application in improving the efficiency of program slicing. *ACM Trans. Softw. Eng. Methodol.*, 11(3):347–383, 2002.

[15] M. Luján, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of java array bounds checks in the presence of indirection. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 76–85, New York, NY, USA, 2002. ACM Press.

[16] T. V. N. Nguyen and F. Irigoin. Efficient and effective array bound checking. *ACM Trans. Program. Lang. Syst.*, 27(3):527–570, 2005.

[17] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society.

[18] N. Suzuki and K. Ishihata. Implementation of an Array Bound Checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.

[19] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[20] Report on the experimental language x10 version 1.01. x10.sf.net/docs/x10-101.pdf.

[21] X10 project on sourceforge. x10.sf.net.