MiX10: Compiling Matlab to X10 for High Performance

Sable Technical Report No. sable-2014-01

Vineet Kumar and Laurie Hendren

March 25, 2014

Contents

List of Figures

Abstract

MATLAB is a popular dynamic array-based language commonly used by students, scientists and engineers who appreciate the interactive development style, the rich set of array operators, the extensive builtin library, and the fact that they do not have to declare static types. Even though these users like to program in MATLAB, their computations are often very compute-intensive and are better suited for emerging high performance computing systems. This paper reports on MIX10, a source-to-source compiler that automatically translates MATLAB programs to X10, a language designed for "Performance and Productivity at Scale"; thus, helping scientific programmers make better use of high performance computing systems.

There is a large semantic gap between the array-based dynamically-typed nature of MATLAB and the object-oriented, statically-typed, and high-level array abstractions of X10. This paper addresses the major challenges that must be overcome to produce sequential X10 code that is competitive with state-of-the-art static compilers for MATLAB which target more conventional imperative languages such as C and Fortran. Given that efficient basis, the paper then provides a translation for the MATLAB `parfor` construct that leverages the powerful concurrency constructs in X10.

The MIX10 compiler has been implemented using the McLab compiler tools, is open source, and is available both for compiler researchers and end-user MATLAB programmers. We have used the implementation to perform many empirical measurements on a set of 17 MATLAB benchmarks. We show that our best MIX10-generated code is significantly faster than the de facto Mathworks' MATLAB system, and that our results are competitive with state-of-the-art static compilers that target C and Fortran. We also show the importance of finding the correct approach to representing the arrays in X10, and the necessity of an *IntegerOkay* analysis that determines which double variables can be safely represented as integers. Finally, we show that our X10-based handling of the MATLAB `parfor` greatly outperforms the de facto MATLAB implementation.

1 Introduction

MATLAB is a popular numeric programming language, used by millions of scientists, engineers as well as students worldwide. MATLAB programmers appreciate the high-level matrix operators, the fact that variables and types do not need to be declared, the large number of library and builtin functions available, and the interactive style of program development available through the IDE and the interpreter-style read-eval-print loop. However, even though MATLAB programmers appreciate all of the features that enable rapid prototyping, their applications are often quite compute intensive and time consuming. These applications could perform much more efficiently if they could be easily ported to a high performance computing system.

One such high performance system is X10, a modern object-oriented and statically-typed language which uses cilk-style arrays indexed by *Point* objects and rail-backed multidimensional arrays, and has been designed with well-defined semantics and high performance computing in mind [7]. The X10 compiler can generate C++ or Java code and supports various communication interfaces including sockets and MPI for communication between nodes on a parallel computing system.

The goal of our research was to develop a bridge between MATLAB and X10, in order to provide MATLAB's ease of use, to benefit from the advantages of static compilation, and to expose scalable concurrency. Our solution is MIX10, an open source-to-source compiler that statically translates MATLAB programs to X10. The MIX10 translator allows scientists and engineers to write programs in MATLAB (or use existing programs already written in MATLAB), and at the same time enjoy the benefits of high performance computing via the X10 system without having to learn a new and

unfamiliar language. Also, since the X10 compiler has back-ends that can produce both C++ and Java, MiX10 can also be used by systems that use MATLAB for prototyping and C++ or Java for production.

This work shares some of the same challenges as other static compilers which convert MATLAB to C or Fortran. However, targeting X10 raises some significant new challenges. For example, X10 is an object-oriented, heap-based, language with varying levels of high-level abstractions for arrays and iterators of arrays. An open question was whether or not it was possible to generate X10 code that both maintains the MATLAB semantics, but also leads to code that is as efficient as state-of-the-art translators that target C and Fortran. Finding an effective and efficient translation from MATLAB to X10 was not obvious, and this paper reports on the key problems we encountered and the solutions that we designed and implemented in our MiX10 system.

By demonstrating that we can generate sequential X10 code that is as efficient as generated C or Fortran code, we then enabled the possibility of leveraging the high performance nature of X10's parallel constructs. To demonstrate this, we examined how to use X10 features to provide a good implementation for MATLAB's `parfor` construct.

The major contributions of this paper are as follows:

Identifying key challenges: We have identified the key challenges in compiling MATLAB to X10.

Techniques for efficiently compiling MATLAB arrays: Arrays are the core of MATLAB. All data, including scalar values are represented as arrays in MATLAB. Efficient compilation of arrays is the key for good performance. We provide techniques to compile MATLAB arrays to two different representations of arrays provided by X10.

Comparison of the two array representations: X10 provides two types of array representations for multidimensional arrays: (1) Cilk-styled, region-based arrays and (2) rail-backed simple arrays. We compare and contrast these two array forms for a high performance computing language in context of being used as a target language.

Safe use of integer variables: Even after determining the correct use of X10 arrays, we were still faced with a performance gap between the code generated by the C/Fortran systems, and the generated X10 code generated by MiX10. Furthermore, we found that the MiX10 system using the X10 Java backend was often generating better code than with the X10 C++ backend, which was counter-intuitive.

We determined that a key remaining problem was overhead due to casting doubles to integers. This casting overhead was quite high, and was substantially higher for the C++ back-end than for the Java back-end (since the casting instructions are handled efficiently by the Java VM). This casting problem arises because the default data type for MATLAB variables is double - even variables used to iterate through arrays, and to index into arrays typically have double type, even though they hold integral values. To tackle this issue we developed an *IntegerOkay* analysis which determines which double variables can be safely converted to integer variables in the generated X10 code, hence greatly reducing the number of casting operations needed.

Code generation strategies for `parfor`: `parfor` allows parallel execution of for loop iterations in MATLAB. We provide technique to effectively compile `parfor` construct to X10.

Open implementation: We have implemented the techniques provided in this paper in an open and extensible framework (`www.sable.mcgill.ca/mclab/mix10.html`). This allows for others to make further improvements to the X10 code generated, or to reuse many of the analyses to generate code for other object-oriented languages.

Experimental evaluation: We have experimented with our MiX10 implementation on a set of benchmarks. Our results show that our generated code is almost an order of magnitude faster than the Mathworks' standard JIT-based system, and is competitive with other state-of-the-art tools that generate C/Fortran. Our more detailed results show the importance of using our customized and optimized X10 array representations, and we demonstrate the effectiveness of the *IntegerOkay* analysis. Finally, we show that our X10-based treatment of `parfor` significantly outperforms MATLAB on our benchmarks.

The remainder of this paper is structured as follows. In Section 2 we describe the background. Section 3 gives an introduction to arrays in X10. In Section 4 we provide our compilation strategies for different array representations and a comparison of both the approaches. Section 5 outlines our new *IntegerOkay* analysis, and Section 6 describes our strategy for handling `parfor`. Section 7 provides an empircal evaluation of our approach. Finally, we provide a discussion of related work in Section 8, and conclude and discuss some planned future work in Section 9.

2 Background

In this section we give the background and context for the MiX10 compiler. Appendix A describes the overall structure of the MiX10 compiler.

MATLAB is actually quite a complicated language to compile, starting with its rather unusual syntax, which cannot be parsed with standard LALR techniques. There are several issues that must be dealt with including distinguishing places where white space and new line characters have syntactic meaning, and filling in missing **end** keywords, which are sometimes optional. The McLAB front-end handles the parsing of MATLAB through a two step process. There is a pre-processing step which translates MATLAB programs to a cleaner subset, called *Natlab*, which has a grammar that can be expressed cleanly for a LALR parser. The McLAB front-end delivers a high-level AST based on this cleaner grammar.

After parsing, the next major phase of MiX10 uses the McSAF framework [4, 3] to disambiguate identifiers using *kind analysis* [5], which determines if an identifier refers to a *variable* or a *named function*. This is required because the syntax of MATLAB does not distinguish between variables and functions. For example, the expression `a(i)` could refer to four different computations, `a` could be an array or a function, and `i` could refer to the builtin function for the imaginary value i, or it could refer to a variable `i`. The McSAF framework also simplifies the AST, producing a lower-level AST which is more amenable to subsequent analysis.

The next major phase is the Tamer [6], which is a key component for any tool which statically compiles MATLAB. The Tamer generates an even more defined AST called *Tamer IR*, as well as performing key interprocedural analyses to determine both the call graph and an estimate of the base type and shape of each variable, at each program point. The call graph is needed to determine which files (functions) need to be compiled, and the type and shape information is very important for generating reasonable code when the target language is statically typed, as is the case for X10.

The Tamer also provides an extensible *interprocedural value analysis* and an interprocedural analysis framework that extends the intraprocedural framework provided by McSaf. Any static backend will use the standard results of the Tamer, but is also likely to implement some target-language-specific analyses which estimate properties useful for generating code in a specific target language. Currently, we have implemented two analyses : (1) An analysis for determining if a Matlab variable is *real* or *complex* to enable support for complex numbers in MiX10 and other Matlab compilers based on McLab; and (2) *IntegerOkay* analysis to identify which variables can be safely declared to be of an integer type (`Int` or `Long`) instead of the default type `Double`.

For the purposes of MiX10, the output of the Tamer is a low-level, well-structured AST, which along with key analysis information about the call graph, the types and shapes of variables, and X10-specific information. These Tamer outputs are provided to the code generator, which generates X10 code, and which is the main focus of this paper.

The X10 source code generator actually gets inputs from two places. It uses the Tamer IR it receives from the the Tamer to drive the code generation, but for expressions referring to built-in Matlab functions it interacts with the *Built-in Handler* which used the built-in template file we provide. We have described the functioning of the built-in handler and a very basic code generation strategy for ordinary sequential constructs at the X10 workshop [11]. In this paper we focus on the challenges in generating *efficient* X10 code whose performance is comparable to state-of-the-art tools that generate more traditional imperative languages like C and Fortran. In this paper we also discuss our strategy for generating parallel X10 code for Matlab `parfor` construct, thus harnessing the high-performance capabilities of X10.

3 Introduction to X10 arrays

In order to understand the challenges of translating Matlab to X10, one must understand the different flavours and functionality of X10 arrays.

At the lowest level of abstraction, X10 provides an intrinsic one-dimensional fixed-size array called `Rail` which is indexed by a `Long` type value starting at `0`. This is the X10 counterpart of built-in arrays in languages like C or Java. In addition, X10 provides two types of more sophisticated array abstractions in packages, `x10.array` and `x10.regionarray`.

Rail-backed Simple arrays are a high-performance abstraction for multidimensional arrays in X10 that support only rectangular dense arrays with zero-based indexing. Also, they support only up to three dimensions (specified statically) and row-major ordering. These restrictions allow effective optimizations on indexing operations on the underlying `Rail`. Essentially, these multidimensional arrays map to a `Rail` of size equal to number of elements in the array, in a row-major order.

Region arrays are much more flexible. A *region* is a set of points of the same rank, where `Points` are the indexing units for arrays. Points are represented as n-dimensional tuples of integer values. The `rank` of a point defines the dimensionality of the array it indexes. The rank of a region is the rank of its underlying points. Regions provide flexibility of shape and indexing. *Region arrays* are just a set of elements with each element mapped to a unique point in the underlying region. The dynamicity of these arrays come at the cost of performance.

Both types of arrays also support distribution across places. A *place* is one of the central innovations in X10, which permits the programmer to deal with notions of locality.

4 Compilation to X10 arrays

Arrays are the core of the MATLAB programming language. Every value in MATLAB is a *Matrix* and has an associated array shape. Even scalar values are represented as 1×1 arrays. Most of the data read and write operations involve accessing individual or a set of array elements. Given the central role of arrays in MATLAB, it is of utmost importance for our MIX10 compiler to find effective and efficient translations to X10 arrays.

Given the shape information provided by the shape analysis engine [13] built into the MCLAB analysis framework [4, 3, 6], it was not hard to compile MATLAB arrays to X10. However, to generate X10 code whose performance would be competitive to the generated C code (via MATLAB coder) and the generated Fortran code (via the MC2FOR compiler), was not straightforward, and required deeper understanding of the X10 array system and careful handling of several features of the MATLAB arrays.

4.1 Simple Arrays or Region Arrays

As described in Section 3, X10 provides two higher level abstractions for arrays, simple arrays, a high performance but rigid abstraction, and region arrays, a flexible abstraction but not as efficient as the simple arrays. In order to achieve more efficiency, our strategy is to use the simple arrays whenever possible, and to fall back to the region arrays when necessary. Note that it is possible to force the MIX10 compiler to use region arrays via a switch, for experimentation purposes.

4.1.1 Compilation to Simple arrays

In dealing with the simple rail-backed arrays, there were two important challenges. First, we needed to determine when it is safe to use the simple rail-backed arrays, and second, we needed an implementation of simple rail-backed arrays that handles the column-major, 1-indexing, and linearization operations required by MATLAB.

When to use simple rail-backed arrays: After the shape analysis of the source MATLAB program, if shapes of all the arrays in the program: (1) are known statically, (2) are supported by the X10 implementation of simple arrays and (3) the dimensionality of the shapes remain same at all points in the program; then MIX10 generates X10 code that uses simple arrays.

Column-major indexing: In order to make X10 simple arrays more compatible with MATLAB, we modified the implementation of the `Array_2` and `Array_3` classes in `x10.array` package to use column-major ordering instead of the default row-major ordering when linearizing multidimensional arrays to the backing `Rail` storage.[1] Since MATLAB uses column-major ordering to linearize arrays, this modification also makes it trivial to support linear indexing operations in MATLAB. [2] MATLAB naturally supports linear indexing for individual element access. More precisely, if the number of subscripts in an array access is less than the number of dimensions of the array, the last subscript is linearly indexed over the remaining number of dimensions in a column-major fashion. Our

[1] `http://www.sable.mcgill.ca/mclab/mix10/x10_update/`

[2] `http://www.mathworks.com/help/matlab/math/matrix-indexing.html`

modification to use column-major ordering for the backing `Rail` make it easier and more efficient to support linear indexing by allowing direct access to the underlying `Rail` at the calculated linear offset.

Given that we can determine when it is safe to use the simple rail-backed arrays, and our improved X10 implementation of them, we then designed the appropriate translations from MATLAB to X10, for array construction, array accesses for both individual elements and ranges. Given the number of dimensions and the size of each dimension, it is easy to construct a simple array. For example a two-dimensional array `A` of type `T` and shape $m \times n$ can be constructed using a statement like `val A:Array_2[T] = new Array_2[T](m,n);`. Additional arguments can be passed to the constructor to initialize the array. Another important thing to note is that MATLAB allows the use of keyword `end` or an expression involving `end` (like `end-1`) as a subscript. `end` denotes the highest index in that dimension. If the highest index is not known the `numElems_i` property of the simple arrays is used to get the number of elements in the `i`th dimension of the array.

4.1.2 Compilation to Region Arays

With MATLAB's dynamic nature and unconventional semantics, it is not always possible to statically determine the shape of an arrays accurately. Luckily, with some thought to a proper translation, X10's region arrays are flexible enough to support MATLAB's "wild" arrays. Also, since `Point` objects can be a set of arbitrary integers, there is no restriction on the starting index of the arrays. Region arrays can easily use one-based indexing.

Array construction: Array construction for region arrays involves creating a region over a set of points (or index objects) and assigning it to an array. Regions of arbitrary ranks can be created dynamically. For example, consider the following MATLAB code snippet:

```
function[x] = foo(a)
    t = bar(a);
    x = t;
    ...
end
```

```
function[y] = bar(a)
    if (a == 3)
        y = zeros(a,a+1,a+2,a+3);
    else
        y = zeros(a,a+1,a+2);
    end
end
```

In this code, the number of dimensions of array `t` and hence array `x` cannot be determined statically at compile-time. In such case, it is not possible to generate X10 code that uses simple arrays, however, it can still be compiled to the following X10 code for function `foo()`.

```
static def foo(a: Double){
  val t: Array[Double] =
    new Array[Double](bar(a));
  val x: Array[Double] =
    new Array[Double](t);
  ...
  return x;
}
```

```
static def bar(a:Double){
  var y:Array[Double]=null;
  if (a == 3) {
    y = new Array[Double]
      (Mix10.zeros(a,a+1,a+2,a+3));
  }
  else {
    y = new Array[Double]
      (Mix10.zeros(a,a+1,a+2));
  }
  return y;
}
```

In this generated X10 code, `t` is an array of type `Double` which can be created by copying from another array returned by `bar(a)` without knowing the shape of the returned array.

Array access: Subscripting operations to access individual elements are mapped to X10's region array subscripting operation. If the rank of array is 4 or less, it is subscripted directly by integers corresponding to subscripts in MATLAB otherwise we create a `Point` object from these integer values and use it to subscript the array. In case an expression involving `end` is used for indexing and the complete shape information is not available, method `max(Long i)`, provided by the `Region` class is used, allowing to determine the highest index for a particular dimension at runtime.

Rank specialization: Although region arrays can be used with minimal compile-time information, providing additional static information can improve performance of the resultant code by eliminating run-time checks involving provided information. One of the key specializations that we introduced with use of region arrays is to specify the rank of an array in its declaration, whenever it is known statically. For example if rank of an array `A` of type `T` is known to be two, it can be declared as `val A:Array[T](2);`. This specialization provided better performance compared to unspecialized code as shown in section 7.6.

4.2 Handling the colon expression

MATLAB allows the use of an expression such as `a:b` (or `colon(a,b)`) to create a vector of integers `[a, a+1, a+2, ... b]`. In another form, an expression like `a:i:b` can be used to specify an integer interval of size `i` between the elements of the resulting vector. Use of a `colon` expression for array subscripting takes all the elements of the array for which the subscript in a particular dimension is in the vector created by the `colon` expression in that dimension.[3] Consider the following MATLAB code:

```
function [x] = crazyArray(a)
    y = ones(3,4,5);
    x = y(1,2:3,:) ;
end
```

Here `y` is a three-dimensional array of shape $3\times4\times5$ and `x` is a sub-array of `y` of shape $1\times2\times5$. Such array accesses can be handled by simply calling the `getSubArray[T]()` that we have implemented in a Helper class provided with the generated code. The generated X10 code with simple array for this example is as follows:

```
static def crazyArray (a: Double){
  val y: Array_3[Double] = new Array_3[Double](Mix10.ones(3, 4, 5));
  val mc_t0: Array_1[Double] = new Array_1[Double](Mix10.colon(2, 3));
  var x: Array_3[Double];
  x = new Array_3[Double](Helper.getSubArray(1, 1, mc_t0(0), mc_t0(1), 1, 5, y)) ;
  return x;
}
```

The colon operator can also be used on the left hand side for an array set operation that updates multiple values of the array. For example, in the MATLAB statement `x(:,4) = y;`, all the values of the fourth column of `x` will be set to `y` if `y` is a scalar or to corresponding values of `y` if `y` is a

[3] Use of ':' in place of an index without lower and upper bounds indicates the use of all the indices in that dimension.

column vector with length equal to the size of first dimension of `x`. To handle this kind of operation we have implemented another helper method, `setSubArray()`. This method takes as input, the target array, the bounds on each dimension, and the source array. `x(:,4) = y;` will be translated by MiX10 to `x = Helper.setSubArray(x, 1, x.numElems_1, 4, 4, y);`

We have implemented overloaded versions of the `getSubArray()` and the `setSubArray()` methods for arrays of different dimensions. For region arrays, we provide the same methods that operate on region arrays in a different version of the Helper class. MiX10 provides the correct version of the Helper class, based on what kind of arrays are used.

4.3 Array growth

Matlab allows explicit array growth during runtime via the `horzcat()` and the `vertcat()` builtin functions for array concatenation operations. In MiX10 this feature is supported for simple arrays as long as the array growth does not change the number of dimensions of the array. For region arrays, this feature is supported in full. For simple arrays, X10 allows a variable declared to be an array of rank i, to hold any array value of the same rank. For example, consider the following set of statements:

```
//...
var x:Array_2[Long];
x = new Array_2(3,4,0);
y = new Array_2(3,5,0);
x=y;
//...
```

Here, `x` is defined to be of type `Array_2[Long]` and can hold arrays of different sizes at different points in the program.

Region arrays, being more dynamic, also support array growth even if it changes the rank of the array. For example, the following set of statements are valid in an X10 program that uses region arrays:

```
//...
var x:Array[Long];
x = new Array(Region.make(1..3,1..4),0);
y = new Array(Region.make(1..3,1..5,1..6),1);
x=y;
//...
```

Here `x` is a 2-dimensional array and `y` is a 3-dimensional array.

Section 7.6 discusses the performance results obtained by using different kinds of arrays and provides a comparison of them, thus showing the efficiency of our approach for compiling Matlab arrays to X10.

5 Safely using integer variables

In this section we present the *IntegerOkay* analysis to identify which variables in the source Matlab program can be safely declared to be of an integer type instead of the default double type. In Matlab all the variables holding a numerical value are by default of type `Double`, which means that by default, in the X10 code generated from Matlab, all variables are statically declared to

be of `Double`. However, in languages like X10, Java and C++, certain program operations require the variables used to be of an integer type. A prominent example of such an operation is an array access operation. An array access requires the variables used to index into the array to be of an integer type. For example, in a statement like `x = A(i,j)`, the variables `i` and `j` are required to be of integer type and result in an error otherwise.

5.1 Need for declaring variables to be of integer type

A simple solution to handle this problem in the generated code from MATLAB is to explicitly cast the variable from `Double` to `Long`, whenever it is required to be used as an integer. However, our experiments showed this approach to be very inefficient. With this approach, we observed that the C++ programs generated by the X10 compiler's C++ backend were slow, and often even slower than the Java code generated by the X10 Java backend for the same program (which was somewhat surprising). The reason for the added slowness in the C++ code was because each typecast from `Double` to `Long` involved an explicit check on the value of the `Double` type variable to ensure that it lies in the 64-bit range supported by `Long`, whereas the cast in Java is handled by a primitive bytecode cast instruction. However, even in Java, extraneous casts clearly hurt performance.

To solve this problem, we designed and implemented the *IntegerOkay* analysis that identifies variables that can be safely declared to be of `Long` type, thus eliminating the need for costly typecasting on these variables.

5.2 An example

To understand the effect on performance caused by typecasting consider a simple example of X10 code shown in listing 1 that just loops over a 2-dimensional array and sets each element `A(i,j)` to `A(i-1,j-1) + A(i+1, j+1)`. In this example, the index variables `i` and `j` are declared to be of type `Double` and are typecast to `Long` when used for indexing into the array. This example reflects the type of X10 code that we would generate if we do not have the *IntegerOkay* analysis.

Listing 2 shows the same example, but with `i` and `j` declared to be `Long`, and thus not requiring an explicit typecast. This example reflects the code that we would be able to generate with a good *IntegerOkay* analysis.

Listing 1: Example for using `Double` variables for array indexing

```
static def useDoubles(scale:Double, n:Long){
  val a: Array_2[Double] =
    new Array_2[Double](Mix10.rand(scale, scale));
  var i:Double = 0; var j:Double = 0; var v:Long = 0;
  for (v=0;v<n;v++){
    for (j=1;j<a.numElems_2-1;j++){
      for (i=1;i<a.numElems_1-1;i++){
        a(i as Long,j as Long) = a(i as Long -1, j as Long -1) +
         a(i as Long +1, j as Long +1);
      } } } }
```

Listing 2: Example for using `Long` variables for indexing

```
static def useLongs(scale:Double, n:Long){
  val a: Array_2[Double] =
    new Array_2[Double](Mix10.rand(scale, scale));
```

```
var i:Long = 0; var j:Long = 0; var v:Long = 0;
for (v=0;v<n;v++){
  for (j=1;j<a.numElems_2−1;j++){
    for (i=1;i<a.numElems_1−1;i++){
      a(i, j) = a(i−1, j−1) + a(i+1, j+1);
    } } } }
```

	input args: 100, 200000		input args : 10000, 20	
	Java	C++	Java	C++
useDoubles	6.9	33.7	7.6	35.2
useLongs	3.4	1.5	3.7	2.0

Table I: Running times (in seconds) for listings 1 and 2, smaller is better

Table I shows running times (in seconds) for these two examples for different values of input arguments. For the listing 1, the C++ code generated by the X10 compiler is nearly 5 times slower as compared to the Java code generated from X10 for the same example. Compared to 2 it is slower than the C++ code for this example by almost 20 times. On the other hand, Java code for the listing 1 is nearly 2 times slower compared to the Java code for the listing 2. For the C++ backend, since the C++ compiler does not provide the checks for `Double` to `Long` typecast, it is implemented in the X10 C++ backend. For the Java backend, X10 relies on these checks provided by the JVM. The more efficient implementation of these checks in the JVM, compared to that in the X10 C++ backend explains for comparatively lower slowdowns for the Java code. Section 7.7 gives detailed evaluation of the performance benefits obtained by using *IntegerOkay* analysis on our benchmark set.

5.3 IntegerOkay Analysis

The basic idea behind the *IntegerOkay* analysis is that, for each variable `x`, if for every use and every definition of `x` in the program `x` can be safely assumed to be an integer, i.e. its declaration as an integer does not change the result of the program, then it can be declared as an integer. Thus, the problem boils down to answering the question of whether each use or a definition, `x` can be safely assumed to be an integer.

There are three possible answers to this question:

1. *IntegerOkay*: The variable use/def can be safely assumed to be an integer. For example, for a definition like `x = 2.0` or for use as an array index like `A(x)`, it is safe to assume that if `x` was declared to be an integer, this definition or use will not affect the result of the program. In other words, for this definition or use of `x`, `x` is `IntegerOkay`.

2. *Not IntegerOkay*: The variable cannot be an integer type. For example consider the expression `x/y`. Here, since the type of the operands can affect the result of the division operation, it is unsafe to assume that `x` and `y` can be of integer type for this particular use. As another example, consider the definition `x = 3.14`. Here, since assuming `x` to be an integer will result in an error, x is not *IntegerOkay*.

3. *Conditionally IntegerOkay*: The variable `x` can be an integer if for the use or definition in question, the variables on which its value depends on, are *IntegerOkay* everywhere in the

program. For example, in a definition like `x = a+b`, `x` can be an integer if both `a` and `b` are integers. We say `x` is conditionally *IntegerOkay* and depends on `a` and `b`. Note that in this particular use of `a` and `b` (as operands of the plus operator), since their type does not affect the result value of the plus operator, `a` and `b` are *IntegerOkay*.

In our MIX10 compiler we solve the *IntegerOkay* problem using a simple fixed-point computation. For each variable use and definition, the algorithm initially associates it with one of the three abstract values above. We then compute the fixed-point by iteratively refining the dependency lists of the conditional variables. Consider each variable x, if every use and definition of x has been determined to be *IntegerOkay*, then x is removed from the dependency lists of all the variables that are *Conditionally IntegerOkay* and depend on x. Once the dependency list for a particular use or definition of a variable is empty, it is upgraded to be *IntegerOkay* for that particular use or definition.

If a variable is not *IntegerOkay* at some point in the program or its dependency list does not become empty for some point in the program (say, for circular dependency), it cannot be declared as an integer. Since, every time we declare a variable to be integer, one or more *Conditionally IntegerOkay* variables might be upgraded to *IntegerOkay*, we iteratively repeat the process of finding variables that are *IntegerOkay* at all points in the program, until we reach a fixed point. Note that since we never downgrade a variable to *Not IntegerOkay* or *Conditionally IntegerOkay*, our iterative algorithm will always terminate.

Consider the following pseudocode for example:

```
/*1*/ x = 3.0;
/*2*/ y = 3.14;
/*3*/ z = x+y;
/*4*/ for (i = 0; i < 5; i++)
/*5*/   y = y+i;
/*6*/ end
```

In this example, the initialization step proceeds as follows. On line 1, `x` is *IntegerOkay* since 3.0 is a *real integer*. On line 2, `y` is *Not IntegerOkay*. On line 3, `z` is *Conditionally IntegerOkay* and depends on `x` and `y`, whereas `x` and `y` are *IntegerOkay* in their use in the expression `x+y`. On line 4, `i` is *IntegerOkay* in its definition `i = 0`, in its use in the expression `i < 5`, and also in the definition `i++`. On line 5, `y` is conditionally *IntegerOkay* and depends on `i` in its definition and it is *IntegerOkay* in its use in `y+i`. `i` is *IntegerOkay* in its use in `y+i`. Note that on line 5, we do not include `y` in its own dependency list, since if we say, `y` is conditionally *IntegerOkay* and depends on `y`, it is safe to declare `y` as integer as long as it does not have any other dependencies and is *IntegerOkay* everywhere else in the program.

The fixed-point solver for this example proceeds as follows. We look for variables that are *IntegerOkay* at every point in the program. `x` and `i` are two such variables and we can declare them to be an integer. We also remove x from the dependency list of definition of `z` on line 3, and `i` from the dependency list of definition of `y` on line 5. Next, we search again and find that `y` is *IntegerOkay* in its use on line 3 and line 5, and also in its definition on line 5, however it is *Not IntegerOkay* in its definition on line 2 and thus it cannot be declared as an integer. `z` on line 3 is dependent on `y` and thus it can also not be declared as an integer. At this point, we have reached a fixed point since there are no more upgrades. Finally, we declare `x` and `i` as integers, and `y` and `z` as doubles.

6 Code generation for the MATLAB `parfor` loop

The MATLAB `parfor` construct is an important feature in MATLAB and is provided by the Mathworks' parallel computing toolbox [16]. It allows the for loop iterations in the MATLAB programs to be executed in parallel, whenever safely possible, thus greatly enhancing the performance of the for loop execution. Other static MATLAB compilers like MATLAB coder and Mc2For do not support the `parfor` loop due to the lack of builtin concurrency features in their target languages, C and Fortran. However, X10, being a parallel programming language, naturally provides concurrency control features. The MiX10 compiler supports parallel code generation for the MATLAB `parfor` construct and provides significantly better performance compared to MATLAB code with `parfor`, and also the sequential version of the X10 code generated for the same program. [4]

The `parfor` (or parallel for loop) is a key parallelization control provided by the MATLAB parallel computing toolbox that can be used to execute each iteration of the for loop in parallel with each other. The challenge was to implement it with X10's concurrency controls while maintaining its complex semantics and aiming for better performance than provided by the parallel computing toolbox. There are three important semantic characteristics of MATLAB's `parfor` loop:

1. the scope of variables inside a `parfor` loop, including the loop index variable, is limited to each iteration.

2. if a variable defined outside the loop is modified inside the loop such that its value after the loop is dependent on the sequence of execution of iterations, then its value after the loop is set to its value before the loop.

3. if a variable defined outside the loop is modified in a reduction assignment i.e., the final value after the iterations is independent of the order of execution of iterations, the updated value is retained after the `parfor` loop. Consider the MATLAB code given on the left of Figure 1.

```
function [] = saneParfor(v)
d = v;
x=0;
A=zeros(1,10);
parfor i = 1:10
   x = x+i;
   d = i*2;
   A(i) = d;
end
disp(d);
end
```

```
static def saneParfor (v: Double)
{ var d: Double = v;
  var x: Double = 0;
  val A: Array_1[Double] =
    new Array_1[Double](Mix10.zeros(1, 10));
  var mc_t3: Double = 1;
  var mc_t4: Double = 10;
  finish {
    for (i in (mc_t3 as Long)..(mc_t4 as Long))
     async {
       atomic x = Mix10.plus(x, i as Double);
       var mc_t2: Double = 2;
       var d_local: Double =
         Mix10.mtimes(i as Double, mc_t2);
       A(i as Long -1) = d_local ;
     }
  }
}
```

Figure 1: Example of `parfor`, MATLAB with `parfor` on the left, generated X10 on the right.

[4] In Appendix B we provide some background on the X10 concurrency features. Readers not familiar with X10 may find it useful to read that appendix before continuing with this section.

Here `x = x+i;` is a reduction assignment [15] statement. The value of `d` is local to each iteration and the initial value before the loop is retained after the loop. Note that the value of `d` outside the loop is invisible inside the loop. For statement `A(i) = d;`, each iteration modifies a unique element accessible only to it, hence the final value of `A` is independent of order of execution; thus its value is updated after the loop.

The MiX10 compiler uses the following strategy to translate `parfor` loops to X10:

1. Introduce `finish` and `async` constructs to control the flow of statements in parallel. This puts the statement immediately after the `for` loop in wait, until all the iterations have been executed.
2. Any variable defined inside the loop and not declared outside the loop previously is declared inside the `async` scope to make it local to the iteration.
3. Any variable defined inside the loop that is previously defined outside the loop and is not a reduction variable is replaced by a local temporary variable defined inside the loop.
4. Statements identified to be reduction statements are made atomic by using the `atomic` construct in X10.

An example of the X10 code generated for the example MATLAB code is given on the right side of Figure 1. The use of `finish` and `async` ensure that each iteration is executed in parallel and the statement after the `for` loop is blocked until all the iterations have finished executing. Note that the `for` loop is iterated over a `LongRange` to ensure that the declaration of the loop variable `i` is local to each iteration. The statement `x = x+i` is a reduction statement, since its order of execution does not affect the value of x at the end of the loop. It is declared to be `atomic` to ensure that the two operations of addition and assignment in the statement are executed as a whole, without any interference from its execution in other iterations. Since the variable `d` is also defined outside the loop, it is replaced by a local variable `d_local` inside the loop. Finally, since each array variable `A(i)` is unique, it is executed normally for each iteration.

To conclude, we can translate the `parfor` in MATLAB to semantically equivalent code in X10 and since X10 can handle massive scaling, we can get significantly better performance for X10 compared to MATLAB as shown by our experimental results in Section 7.8.

7 Evaluation

In this section we evaluate the performance of our compiler. In this research our main aim was to generate X10 code for MATLAB such that its sequential performance would be comparable to the performance provided by the state of the art tools which translate MATLAB to more traditional imperative languages such as C and Fortran. To demonstrate our results, we compiled a set of 17 MATLAB programs to X10 via the MiX10 compiler and compared their performance results with those of the original MATLAB programs, C programs generated for our benchmarks via the MATLAB coder, and Fortran programs generated by the Mc2For compiler.[5] In addition to showing our best

[5]We also compared our results to Octave, a widely used open source alternative to MATLAB. However, since Octave involves an interpreter, it performed slower than the standard MATLAB compiler (with a mean slowdown of 66.67 times slower) over all of our benchmarks, thus in this section we do not concentrate on comparison of our results with Octave.

Benchmark	Source	Description	Key features
bbai	MATLAB file exchange	Implementation of the Babai estimation algorithm	2-D arrays, random number generation
bubl	McLab	Bubble sort	1-D array, nested loops
capr	Chalmers University	Computes the capacitance of a transmission line using finite difference and Gauss-Seidel method	Array operations on 2-D arrays, nested loops
clos	Otter project	Calculates the transitive closure of a directed graph	Matrix multiplication, 2-D arrays
crni	Falcon project	Crank-Nicholson solution to the heat equation	read/write operations on a very large 2-D array
dich	Falcon project	Dirichlet solution to Laplace's equation	Array operations on 2-D arrays, nested loops
diff	MATLAB file exchange	Calculates the diffraction pattern of monochromatic light	2-D arrays, Concatenation operations, complex numbers
edit	MATLAB file exchange	Calculates the edit distance between two strings	many 1-D arrays of characters
fiff	Falcon project	Computes the finite difference solution to the wave equation	Array operations on 2-D arrays, nested loops
lgdr		Calculates derivatives of Legendre polynomials	Array transpose on row vectors
mbrt	MCFOR project	Computes Mandelbrot sets	Complex numbers, `parfor` loop
nb1d	Otter project	Simulates the 1-dimensional n-body problem	Column-vectors, nested loops, `parfor` loop
matmul	McLab	naive matrix multiplication	2-D arrays, nested loops, `parfor` loop
mcpi	McLab	Calculates π by the Monte Carlo method	Scalar values, Random number generation, `parfor` loop
numprime	Burkardt and Cliff	Simulates the sieve of Eratosthenes for calculating number of prime numbers less than a given number	Scalar values, nested loops, `parfor` loop
optstop	Burkardt and Cliff	Solution to the optimal stopping problem	Row vectors, random number generation, `parfor` loop
quadrature	Burkardt and Cliff	Simulates the quadrature approach for calculating integral of a function	Scalar values, `parfor` loop

Table II: Benchmarks

overall sequential performance, we also demonstrate the results of compiling the generated X10 code to Java compared to C++, effects on the performance for the various efficiency enhancing techniques discussed in this paper, and finally the performance of the parallel X10 code generated for MATLAB `parfor` loops.

7.1 Benchmarks

The set of benchmarks used for our experiments consists of benchmarks from various sources; Most of them are from related projects like FALCON [19] and OTTER [18], Chalmers university of Technology[6], "Mathworks' central file excahange"[7], and the presentation on parallel programming in MATLAB by Burkardt and Cliff[8]. This set of benchmarks covers the commonly used MATLAB features like arrays of different dimensions, loops, use of numerical functions like random number generation, trigonometric operations, and array operations like transpose and matrix multiplication. Table II gives a description of all the benchmarks we used and shows their special features.

[6] `http://www.elmagn.chalmers.se/courses/CEM/`

[7] `http://www.mathworks.com/matlabcentral/fileexchange`

[8] `http://people.sc.fsu.edu/~jburkardt/presentations/matlab_parallel.pdf`

7.2 Experimental setup

We used Mathworks' MATLAB release R2013a to execute our benchmarks in MATLAB and MATLAB coder. We also executed them using the GNU Octave version 3.2.4. We compiled our benchmarks to Fortran using the Mc2FOR compiler and compiled the generated Fortran code using the GCC 4.6.3 GFortran compiler with optimization level `-O3`. To compile the generated X10 code from our MIX10 compiler, we used X10 version 2.4.0. We used OpenJDK Java 1.7.0_51 to compile and run Java code generated by the X10 compiler, and GCC 4.6.4 g++ compiler to compile the C++ code generated by the X10 compiler. All the experiments were run on a machine with Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz processor and 16 GB memory running GNU/Linux(3.8.0-35-generic #52-Ubuntu). For each benchmark, we used an input size to make the program run for approximately 20 seconds on the de facto MATLAB compiler. We used the same input sizes for compiling and running benchmarks via other compilers. We collected the execution times (averaged over five runs) for each benchmark and compared their speedups over Mathworks' MATLAB runtimes (normalized to one).

7.3 X10 Compiler variations

The MIX10 compiler compiles the source MATLAB code to X10 code, which is then compiled by the X10 compiler. The X10 compiler is also a source to source compiler that provides two backends, a C++ backend that generates C++ code and a Java backend that generates Java code, which are then compiled by their respective compilers to executable code. Both these backends provide a `-NO_CHECKS` switch that generates the C++/Java code that does not include dynamic array bounds checks, which are otherwise included by default. As we described in section 4.1.1, we altered the X10 compiler to use column-major array indexing. We always used the `-O` optimization flag for the X10 compiler for both the backends, with notable exceptions where the X10 optimizer generated code which interacted extremely negatively with the Java JIT, as discussed in section 7.5.1. For all of our experiments, we used our IntegerOkay analysis, except for the experiment which investigates the performance impact of this analysis. Our best results were obtained by compiling the generated X10 code with the C++ backend with `-NO_CHECKS` enabled, where the X10 code itself was generated by the MIX10 compiler with simple arrays and our IntegerOkay analysis enabled.

7.4 Overall MIX10 performance

We compared the performance of the generated X10 code with that of the original MATLAB code run on Mathworks' implementation of MATLAB. To compare against the state of the art static compilers, we also compared the performance of the MIX10 generated X10 code with the C code generated by MATLAB coder and the Fortran code generated by the Mc2FOR compiler.

Figure 2 shows the speedups and slowdowns for the code generated for our benchmarks by different compilers. For MIX10 we have included the results for the X10 code compiled by the X10 C++ backend compiled, once with `-NO_CHECKS` enabled and once with `-NO_CHEKCS` disabled. For Fortran we included the results for the code generated without bounds checks. C code from MATLAB coder was generated with default settings and includes bounds checks. We also calculated the geometric mean of speedups(slowdowns) for all the benchmarks for each compiler.

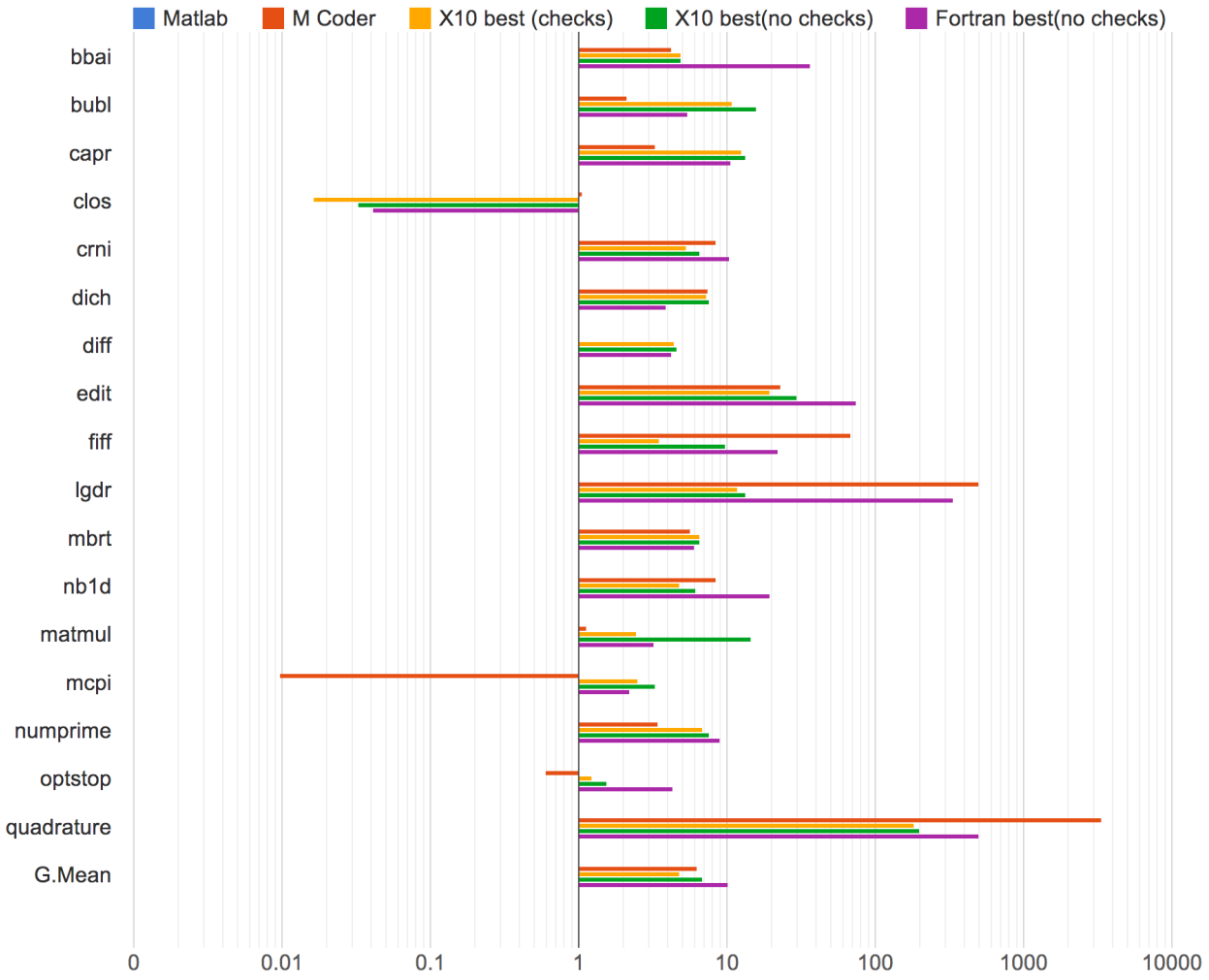

Figure 2: Performance of MiX10 vs other state-of-the-art static compilers, reported as speedups relative to Mathworks' MATLAB, higher is better.

Overall, one can see that the static compilers all provide excellent speedups, often of at least an order of magnitude faster. Thus, for the kinds of benchmarks in our benchmark set, it would seem that tools like the MATLAB coder, Mc2For, and our MiX10 tools are very useful. MiX10 outperforms MATLAB coder in 9 out of 17 benchmarks, when compared with the X10 version compiled with bounds checks, and 10 out of 17 benchmarks when compared with the version with no bounds checks. For Fortran, the generated X10 does better in 7 out of 17 benchmarks with no bounds checks, and 6 out of 17 benchmarks with bounds checks enabled. Note that MATLAB coder was not able to compile 1 of our benchmarks (*diff*) due to the dynamic array growth involved in it; MiX10 supports dynamic array growth.

We achieved a mean speedup of 4.8 over MATLAB, for the X10 code with bounds checks, and 6.8 for the x10 code with no bounds checks. On the other hand, MATLAB coder gave a mean speedup of 6.3 and Mc2For gave a mean speedup of 10.2. However, we noticed that our mean result was skewed due to two benchmarks for which the generated X10 performed very poorly compared to the generated C code. These benchmarks are *clos* and *lgdr*. In the following paragraphs we explain the reason for their poor performance. If we do not consider these two benchmarks, we get a mean

speedup of 6.7 for the X10 code with bounds checks compared to 5.2 for the C code. For the X10 code compiled with no bounds checks we get a mean speedup of 9.3 compared to 11.6 for Fortran.

clos involves repeated calls to the builtin matrix multiplication operation for the 2-dimensional matrices. The generated C code from MATLAB coder uses highly optimized matrix multiplication libraries compared to the naive matrix multiplication implementation used by MIX10. Thus, MIX10 gets a speedup of 0.02 as compared to 1.05 for C. Note that the generated Fortran code is also slowed down (speedup of 0.04) due to the same reason. As a future work, We plan to replace our matrix multiplication implementation with calls to an optimized library function.

lgdr involves repeated transpose of a row vector to a column vector. MATLAB and Fortran, both being array languages are highly optimized for transpose operations. MIX10 currently uses a naive transpose algorithm which is not optimized. We did achieve a speedup of over 10 times compared to MATLAB but it is not as good as the speedups achieved by C (speedup of 505.0) and Fortran (speedup of 336.7). For transpose operation also, we plan to replace our current implementation with an optimized implementation or a call to an optimized library function.

Both of these examples show that in addition to generating good code, another important task is developing optimized X10 library routines for key array computations.

Other interesting numbers are shown by *optstop*, *fiff*, *nb1d*, and *quadrature*. *optstop* gave a speedup of just 1.5 even without bounds checks. It involves repeated random number generation, which our experiments showed to be slow for the X10 C++ backend compared to Fortran and even the Java backend. This problem is worse with C, due to which the C code from MATLAB coder gives a speedup of mere 0.6(slowdown). Fortran performs better with a speedup of 4.3. *bbai* shows a similar pattern due to the same reason. *fiff* is characterized by stencil operations in a loop on a 2-dimensional array. These operations are also optimized by array-based languages like Fortran and MATLAB. For *nb1d*, Fortran performs better due to the use of column vectors in the benchmark, which are represented as 2-dimensional arrays in X10 but in Fortran they are represented as 1-dimensional and are optimized. 2-dimensional arrays are not as fast in X10 as the 1-dimensional arrays. *quadrature* involves repeated arithmetic calculations on a range of numbers. We achieve a speedup of about 200 times compared to MATLAB, however it is slow compared to speedups of 3348 and 502 by C and Fortran respectively. We believe that MATLAB coder leverages partial evaluation for optimizing numerical methods' implementations.

For most of the other benchmarks, we perform better or nearly equal to C and Fortran code. Despite the facts that: (1) the sequential core of X10 is a high-level object oriented language which is not specialized for array-based operations; and (2) Generating the executable binaries via MIX10 involves two levels of source-to-source compilations (MATLAB $\rightarrow$ X10 $\rightarrow$ C++); we have achieved performance comparable to C, the state of the art in statically compiled languages and Fortran, a statically compiled language highly specialized for arrays.

7.5 X10 C++ backend vs. X10 Java backend

The X10 compiler provides two backends, a C++ backend that compiles the X10 code to native binary via C++, and a Java backend that compiles the X10 code to JVM code via Java. We were interested to see how well these backends perform when used to compile the MIX10 generated code. Even though we did not expect Java code to perform as well as the C++ code, our aim was to make sure that we achieved good performance, significantly better than MATLAB, for the X10

Java backend. This would enable MATLAB programmers to use our MiX10 compiler to generate code that could be integrated into Java applications.

In this subsection we present the performance comparison of the MiX10 generated X10 code compiled by the X10 C++ backend with that compiled by the X10 Java backend. Columns 3 and 4 of the Table III show speedups for our benchmarks compiled with the X10 C++ backend without bounds checks, and with bounds checks respectively. Columns 5 and 6 show these values for compilation with the X10 Java backend without bounds checks, and with bounds checks respectively. We also show the geometric mean of the speedups for all the 4 cases.

Benchmark	Matlab	C++ (no checks)	C++ (checks)	Java (no checks)	Java (checks)
bbai	1	4.9	4.9	11.3	10.7
bubl	1	15.8	10.8	7.5	7.5
capr	1	13.5	12.7	11.1	6.3
clos	1	0.03	0.02	0.02	0.002
crni	1	6.5	5.3	5.6	4
dich	1	7.6	7.3	7	1.6
diff	1	4.6	4.4	0.3	0.3
edit	1	29.7	19.4	22.1	20
fiff	1	9.8	3.5	2.1	1.4
lgdr	1	13.5	11.9	10.6	10.6
mbrt	1	6.5	6.5	0.3	0.3
nb1d	1	6.2	4.8	5.5	4.1
matmul	1	14.7	2.5	1.1	0.8
mcpi	1	3.3	2.5	2.9	3
numprime	1	7.6	6.8	6.5	6.4
optstop	1	1.5	1.2	1.8	1.4
quadrature	1	200.9	182.6	167.4	154.5
Geometric mean	1	6.8	4.8	3.4	2.4

Table III: MiX10 performance comparison : X10 C++ backend vs. X10 Java backend, speedups relative to Mathworks' MATLAB, higher is better

The mean speedups for the C++ backend are 6.8 and 4.8 respectively for the version with bounds checks switched off, and the version with bounds checks switched on, whereas for Java backend these values are 3.4 and 2.4 respectively. This is expected, given that C++ is compiled to native binary while Java is JIT compiled.

bbai and *optstop* are the two exceptions, where Java performs better than C++. For *bbai*, both the Java versions gave a speedup of over 10, whereas for C++, the speedup is under 5 for both the versions. For *optstop*, the difference is not large, with C++ speedup at 1.5 (1.2 for the bounds checks version) compared to 1.8 (1.4 for the bounds checks version) for the Java backend. *bbai* is slower with X10 C++ backend because it includes repeated calls to the X10's `Random.nextDouble()` function to generate random numbers. We found it to be significantly slower in the C++ backend compared to the Java backend. We have reported our findings to the X10 development team and they have validated our findings. *optstop* is slower for the same reason : It also involves repeated random number generation. Note that for these two benchmarks, even the C code generated via MATLAB coder is slower than the C++ code, with speedups of 4.2 and 0.6 respectively for *bbai* and *optstop*.

Other interesting results are for the benchmarks *diff*, *fiff*, *mbrt* and *matmul*. For these benchmarks, results from the Java backend are significantly slower compared to the C++ backend. *diff* and

mbrt involve operations on complex numbers. In the X10 C++ backend, complex numbers are stored as `structs` and are kept on the stack, whereas in the Java backend, they are stored as *objects* and reside in the heap storage. *fiff* and *matmul* are characterized by repeated array access and read/write operations on 2-dimensional arrays. For these benchmarks, the Java backend performs significantly slower compared to the C++ backend with no bounds checks(2.1 vs. 9.8 for *fiff* and 1.1 vs. 14.7 for *matmul*), however compared to the performance by the C++ beckend with bounds checks, it is not as slow(2.1 vs. 3.5 for *fiff* and 1.1 vs. 2.5 for *matmul*). The reason is that even with bounds checks turned off for the X10 to Java compiler, The Java compiler by default has bounds checks on. These checks have a significant effect on performance for 2-dimensional array operations.

7.5.1 When not to use the X10 `-O`

One of the most surprising results in this set of experiments was the fact that we had to sometimes disable the X10 `-O` optimizer switch when using the X10 Java backend.. For the benchmarks *capr* and *dich*, in the case when X10 bounds checks are switched on, we found very pathological performance, with slowdowns of over 2 orders of magntitude, when the X10 compiler's optimization switch(`-O`) was used.

We recorded running times of 785.3 seconds for *capr* compared to 3.2 seconds without the optimization, and 1558.9 seconds for *dich* compared to 12.7 seconds without the optimization. With the help of the X10 development team we determined that switching on the optimization triggered code inlining for the array bounds check code, which then caused the resultant Java program to be too large to be handled by the JIT compiler. In fact, the Java JIT effectively gives up on this code and reverts to the interpreter.

Thus, it would seem that the X10 optimizer needs to be improved in order to apply aggressive inlining only when it does not have a negative impact on code size, and that different inlining strategies are needed for the C++ and Java backends.

7.6 Simple vs. Region arrays

One of the key optimizations used by MIX10 is to use simple arrays, wherever possible, for higher performance. In this subsection we discuss the performance gains obtained by using simple arrays over region arrays and the specialized region arrays. A description of these three kinds of arrays provided by X10 was given in Section 3. Table IV shows the relative speedups and slowdowns for our benchmarks compiled to use different kinds of X10 arrays for the C++ backend and the Java backend.

For the C++ backend we obtained a mean speedup of 6.8 for Simple arrays, compared to 2.7 for region arrays and 2.8 for specialized region arrays. For the Java backend, we obtained speedups of 3.4, 1.3 and 1.9 for the simple arrays, region arrays, and the specialized region arrays respectively. These results are as expected in Section 4. For the C++ backend, most noticeable performance differences between simple arrays and region arrays are for *edit*, *fiff*, *lgdr*, *nb1d*, *matmul* and *optstop*. All of these benchmarks are characterized by large number of array accesses and read/write operations on 2-dimensional arrays, except *optstop* and *edit*, which have multiple large 1-dimensional arrays. The performance difference is most noticeable for *nb1d*, where the region arrays are about 20 times slower than the simple arrays. This is because *nb1d* involves simple operations on a large

		X10 C++ backend			X10 Java backend		
Benchmark	Matlab	Simple arrays	Region arrays	Special region arrays	Simple arrays	Region arrays	Special regio
bbai	1.0	4.9	2.7	2.7	11.3	6.4	6.6
bubl	1.0	15.8	11.4	11.6	7.5	3.6	3.7
capr	1.0	13.5	11.6	12.4	11.1	0.02	10.5
clos	1.0	0.03	0.01	0.01	0.02	0.01	0.01
crni	1.0	6.5	3.6	3.9	5.6	4	4.1
dich	1.0	7.6	6.7	7.0	7.0	0.01	0.02
diff	1.0	4.6	4.2	4.3	0.3	0.3	0.3
edit	1.0	29.7	4.3	3.6	22.1	9.4	9.4
fiff	1.0	9.8	2.0	2.8	2.1	1.4	1.4
lgdr	1.0	13.5	1.6	1.5	10.6	5.4	5.4
mbrt	1.0	6.5	6.3	6.5	0.3	5.1	5.1
nb1d	1.0	6.2	0.3	0.3	5.5	1.4	1.4
matmul	1.0	14.7	1.3	1.4	1.1	0.5	0.5
mcpi	1.0	3.3	2.8	2.7	2.9	3.0	3.0
numprime	1.0	7.6	5.7	5.7	6.5	6.3	6.3
optstop	1.0	1.5	0.4	0.4	1.8	1.2	1.3
quadrature	1.0	200.9	200.9	200.9	167.4	143.5	154.5
Geometric mean	1.0	6.8	2.7	2.8	3.4	1.3	1.9

Table IV: MiX10 performance comparison : Simple arrays vs. Region arrays vs. Specialized region arrays, speedup relative to Mathworks' MATLAB, higher is better

column vector. With simple arrays, since the compiler knows that it is a column vector, rather than a 2-dimensional matrix, even though it is declared as a 2-dimensional array, the performance can be optimized to match that of the underlying `Rail`. However, this is not possible for region arrays where the size of each dimension is not known statically. For the C++ backend, we do not observe significant performance differences between the region arrays and the specialized region arrays.

For the Java backend we observed a higher difference in the mean performance for the simple arrays and the region arrays. The mean speedup for region arrays is 1.3 whereas for simple arrays it is nearly 3 times more at 3.4. There is also a significant difference between the performance of specialized region arrays and region arrays. Speedup for specialized region arrays is 1.9. Like the C++ backend, here also, most noticeable performance gain for simple arrays is for benchmarks involving a large number of array accesses and read/writes. *capr* and *dich* ask for special consideration. For *capr*, even with X10 compiler's bounds checks turned off, the region array version slows down by more than 500 times compared to the simple array version and even the specialized region array version. This again, is due to the fact that region arrays, with the dynamic shape checks, generated more code than the JIT compiler could handle. For *dich* the slowdown due to region arrays was about 700 times compared to simple arrays. For *dich*, even the specialization on region arrays was not enough to reduce the code size enough to be able to be JIT compiled.

7.7 Effect of IntegerOkay analysis

In this subsection we present an overview of the performance improvements achieved by MiX10 by using the IntegerOkay analysis. Table V shows a comparison of speedups gained by using the IntegerOkay analysis over those without using it. For this experiment we used results with X10 optimizations turned on and bounds checks turned off.

For the C++ backend, we observed a mean speedup of 6.8 which is two times the speedup gained by not using the IntegerOkay analysis, which is equal to 3.4. We observed a significant gain in

		X10 C++ backend		X10 Java backend	
Benchmark	Matlab	IntegerOkay	All Doubles	IntegerOkay	All Doubles
bbai	1.0	4.9	3.8	11.3	8.2
bubl	1.0	15.8	2.2	7.5	4.2
capr	1.0	13.5	1.7	11.1	9.7
clos	1.0	0.03	0.02	0.02	0.01
crni	1.0	6.5	2.8	5.6	5.5
dich	1.0	7.6	1.0	7.0	5.8
diff	1.0	4.6	4.5	0.3	0.3
edit	1.0	29.7	13.5	22.1	20.0
fiff	1.0	9.8	1.0	2.1	1.8
lgdr	1.0	13.5	10.1	10.6	11.0
mbrt	1.0	6.5	6.1	0.3	0.3
nb1d	1.0	6.2	5.2	5.5	5.5
matmul	1.0	14.7	14.5	1.1	1.2
mcpi	1.0	3.3	3.3	2.9	2.9
numprime	1.0	7.6	7.6	6.5	7.1
optstop	1.0	1.5	1.0	1.8	1.0
quadrature	1.0	200.9	182.6	167.4	143.5
Geometric mean	1.0	6.8	3.4	3.4	2.9

Table V: Performance evaluation for the IntegerOkay analysis, speedups relative to Mathworks' MATLAB, higher is better

performance by using IntegerOkay analysis for the benchmarks that involve significant number of array indexing operations. *bubl*, *capr*, *crni*, *dich*, and *fiff* show the most significant performance gains. The reason for this behaviour is that, X10 requires all array indices to be of type `Long`, thus if the variables used as array indices are declared to be of type `Double` (which is the default in MATLAB), they must be typecast to `Long` type. `Double` to `Long` is very time consuming because every cast involves a check on the value of the `Double` type variable to ensure that it can safely fit into `Long` type.

For the Java backend, with the IntegerOkay analysis, we get a mean speedup of 3.4 as compared to 2.9 without it. The reason for the lower difference as compared to that for the C++ backend is that, for Java backend, the X10 compiler does not generate the value checks for `Double` type values, instead it relies on the JVM to make these checks, resulting in a better performance. Also note that, without the IntegerOkay analysis, the C++ backend results are slower than the Java backend results for 9 out of 17 benchmarks.

To conclude, IntegerOkay analysis is very important for efficient performance of code involving Arrays, specially for the X10 C++ backend.

7.8 MATLAB parfor vs. MiX10 parallel code

Given that we have established that we can generate competitive sequential code, we wanted to also do a preliminary study to see if we could get additional benefits from the high-perfomance nature of X10. In this subsection we present the preliminary results for the compilation of MATLAB `parfor` construct to the parallel X10 code. 7 out of our 17 benchmarks could be safely modified to use the `parfor` loop. We compare the performance of the generated parallel X10 programs for these benchmarks to that of MATLAB code using `parfor`, and to their sequential X10 versions. For this experiment, we used the sequential versions of the generated X10 programs with optimizations and no bounds checks. For the parallel versions we used both the variants, with optimizations and

bounds checks, and with optimizations and without bounds checks. Table VI shows the results for both the X10 C++ and the X10 Java backends.

			X10 C++ backend			X10 Java backend		
			Sequential	Parallel		Sequential	Parallel	
Benchmarks	MATLAB	MATLAB parfor	No checks	No checks	Checks	No checks	No checks	Checks
mbrt	1.0	1.7	6.5	25.3	25.3	0.3	1.3	1.3
nb1d	1.0	0.4	6.2	7.3	7.3	5.5	19.0	15.1
matmul	1.0	1.3	14.7	137.5	3.9	1.1	1.1	0.8
mcpi	1.0	4.6	3.3	15.7	15.9	2.9	18.1	18.2
numprime	1.0	5.3	7.6	30.0	30.9	6.5	24.8	26.4
optstop	1.0	4.1	1.4	2.0	2.1	1.8	10.7	9.7
quadrature	1.0	3.8	200.9	11.3	11.2	167.4	13.0	13.0
Geometric mean	1.0	2.3	8.9	16.0	9.8	3.7	7.8	7.1

Table VI: Performance evaluation for MiX10 generated parallel X10 code for the MATLAB `parfor` construct, speedups relative to Mathworks' MATLAB, higher is better

For the X10 C++ backend we achieved a mean speedup of 16.0 for the generated parallel X10 code without bounds checks, which is over 9 times of the speedup for the MATLAB code with `parfor`, at a speedup of 2.3. Compared to X10 sequential code which has a mean speedup of 8.9, it is nearly twice as fast. Even with the parallel X10 code with bounds checks we achieved a speedup of 9.8 which is over 4 times better than the MATLAB `parfor` code. *optstop* is an interesting exception. It is actually slower (at a speedup of 2.1) than the MATLAB `parfor` version (at a speedup of 4.1). The sequential version of *optstop* is just slightly faster than the sequential MATLAB version, with a speedup of just 1.5 (due to the reasons explained earlier) and the total time for the parallel execution of each iteration, and managing the parallelization of these activities is just slightly faster than the sequential version. We see a similar trend for the *matmul* benchmark for the X10 Java backend. It shows a speedup of just 0.8 (version with bounds checks) as compared to 1.3 for the MATLAB `parfor` version. Overall, for the Java backend we obtained a mean speedup of 7.8 for the X10 code with no bounds checks and 7.1 for the code with bounds checks. Compared to the MATLAB parfor version we obtained about 3.5 times better performance. The parallel X10 code is over 2 times faster than the sequential X10 code (speedup of 3.7).For both, the C++ backend, and the Java backend, the mean speedup for the sequential X10 code is also substantially faster than the MATLAB parallel code.

To conclude, the parallel X10 code provides much higher performance gains compared to the MATLAB `parfor` code and even the X10 sequential code, which by itself is most of the times faster than the MATLAB parallel code.

7.9 Conclusion

We showed that the MiX10 compiler with its efficient handling of array operations and optimizations like IntegerOkay can generate X10 code that provides performance comparable to the native code generated by the state of the art languages like C, which is fairly low-level, and Fortran, which itself is an array-based language. As a future work, we plan to use more efficient implementations of the builtin functions, and believe that it would further improve the performance of the code generated by MiX10.

With MiX10, we also took first steps to compile MATLAB to parallel X10 code to take full advantage of the high performance features of X10. Our preliminary results are very inspiring and we plan to continue in this direction further, in the future.

In addition to demonstrating that our approach leads to good code, these experiments have also been quite valuable for the X10 development team. Our generated code has stressed the X10 system more than the hand-written X10 benchmarks and has exposed places where further improvements can be made.

8 Related Work

The work presented in this paper provides an alternative to Mathworks' de facto proprietary implementation of MATLAB. Our approach is open and extensible and leverages the high-performance computing abilities of X10.

Although our focus is on handling MATLAB itself, notable open source alternatives of MATLAB like Scilab[10], Julia[2], NumPy[20] and Octave[1] provide limited concurrency features. They concentrate on providing open library support and have not tackled the problems of static compilation. We are investigating if there is any way of sharing some of their library support with MiX10. The MEGHA project[17] provides an interesting approach to map MATLAB array operations to CPUs and GPUs, but supports only a very small subset of MATLAB.

There have been previous research projects on static compilation of MATLAB which focused particularly on the array-based subset of MATLAB and developed advanced static analyses for determining shapes and sizes of arrays. For example, FALCON [19] is a MATLAB to FORTRAN90 translator with sophisticated type inference algorithms. The McLab group has previously implemented a prototype Fortran 95 generator [12], and has more recently developed the next generation Fortran generator, Mc2For [13] in parallel with the MiX10 project. Some of the solutions are shared between the projects, especially the parts which extend the Tamer.

MATLAB Coder is a commercial compiler by MathWorks[14], which produces C code for a subset of MATLAB.

In terms of source-to-source compilers for X10, we are aware of two other projects. StreamX10 is a stream programming framework based on X10 [22]. StreamX10 includes a compiler which translates programs in COStream to parallel X10 code. Tetsu discusses the design of a Ruby-based DSL for parallel programming that is compiled to X10 [21].

9 Conclusions and Future Work

This paper is about providing a bridge between two communities, the scientists/engineers/students who like to program in MATLAB on one side; and the programming language and compiler community who have designed elegant languages and powerful compiler techniques on the other side.

The X10 language has been designed to provide high-level array and concurrency abstractions, and our main goal was to develop a tool that would allow programmers to automatically convert their MATLAB code to efficient X10 code. In this way programmers can port their existing MATLAB

code to X10, or continue to develop in MATLAB and use our MiX10 compiler as a backend to generate X10 code. Since X10 is publicly available under the Eclipse Public License (`x10-lang.org/home/x10-license.html`), users could have efficient high-performance code that they could freely distribute. Further, X10 itself can compile the code to either Java or C++, so our tool could be used in a tool chain to convert MATLAB to those languages as well.

Our tool is part of the McLab project, which is entirely open source. Thus, we are providing an infrastructure for other compiler researchers to build upon this work, or to use some of our approaches to handle other popular languages such as R.

In this paper we demonstrated the end-to-end organization of the MiX10 tool, and we identified that the correct handling of arrays, the minimization of casting by safely mapping MATLAB double variables to X10 integer variables via *IntegerOkay* analysis, and concurrency features were the key challenges. We developed a custom version of X10's rail-backed simple arrays, and identified where and how this could be used for generating efficient X10 code. For cases where precise static array shape and type information is not available, we showed how we can use the very flexible region-based arrays in X10, and our experiments demonstrated that it is very important to use the simple rail-backed arrays, for both the Java and C++ backends for X10.

Our experiments show that our generated X10 code is competitive with other state-of-the art code generators which target more traditional languages like C and Fortran. The C++ X10 backend produces faster code than the Java backend, but good performance is achieved in both cases.

One of the main motivations of choosing X10 as the target language is that it supports high-performance computing, which is often desirable for the computation-intensive applications developed by the engineers and scientists. To demonstrate how to take advantage of X10's concurrency, we presented an effective translation of the MATLAB `parfor` construct to semantically equivalent X10.

Our experiments showed significant performance gains for our generated parallel X10 code, as compared to MATLAB's parallel toolbox. This confirms that compiling MATLAB to a modern high-performance language can lead to significant performance improvements.

Based on our positive experiences to date, we plan to continue improving the MiX10 tool. The code that we generate is already quite clean, but we would like to apply further transformations on it to aggregate some low-level expressions, and to make the generated code look as "natural" as possible. We also would like to experiment further to find the best way to tune the generated code for different sorts of parallel architectures. Our experiments also show that there are some key library routines such as matrix multiply and transpose for which we need to have better X10 code. Thus, there is scope for more work on X10 libraries, which could be useful for other source-to-source compilers targeting X10.

Our tool is open source, and we hope that the other research teams will use our infrastructure, as well as learn from our experiences of generating effective and efficient X10 code.

References

[1] GNU Octave. `http://www.gnu.org/software/octave/index.html`.

[2] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A Fast Dynamic Language for Technical Computing. *CoRR*, abs/1209.5145, 2012.

[3] J. Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[4] J. Doherty and L. Hendren. McSAF: A static analysis framework for MATLAB. In *Proceedings of ECOOP 2012*, pages 132–155, 2012.

[5] J. Doherty, L. Hendren, and S. Radpour. Kind analysis for MATLAB. In *In Proceedings of OOPSLA 2011*, pages 99–118, 2011.

[6] A. Dubrau and L. Hendren. Taming MATLAB. In *Proceedings of OOPSLA 2012*, pages 503–522, 2012.

[7] IBM. X10 programming language. `http://x10-lang.org`, Feb. 2012.

[8] IBM. APGAS programming in X10 2.4, 2013. `http://x10-lang.org/documentation/tutorials/apgas-programming-in-x10-24.html`.

[9] IBM. An introduction to X10, 2013. `http://x10.sourceforge.net/documentation/intro/latest/html/node4.html`.

[10] INRIA. Scilab, 2009. `http://www.scilab.org/platform/`.

[11] V. Kumar and L. Hendren. First steps to compiling Matlab to X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 2–11, New York, NY, USA, 2013. ACM.

[12] J. Li. McFor: A MATLAB to FORTRAN 95 Compiler. Master's thesis, McGill University, August 2009.

[13] X. Li and L. Hendren. Mc2for: A tool for automatically translating matlab to fortran 95. In *In Proceedings of CSMR-WCRE 2014*, pages 234–243, 2014.

[14] MathWorks. MATLAB Coder. `http://www.mathworks.com/products/matlab-coder/`.

[15] Mathworks. Reduction variables. `http://www.mathworks.com/help/distcomp/advanced-topics.html#bq_of7_-3`.

[16] MathWorks. Parallel computing toolbox, 2013. `http://www.mathworks.com/products/parallel-computing/`.

[17] A. Prasad, J. Anantpur, and R. Govindarajan. Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 152–163, New York, NY, USA, 2011. ACM.

[18] M. J. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary Results from a Parallel MATLAB Compiler. In *Proceedings of Int. Parallel Processing Symp.*, IPPS, pages 81–87, 1998.

[19] 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.

[20] Scipy.org. Numpy. `http://www.numpy.org/`.

[21] T. Soh. Design and implementation of a DSL based on Ruby for parallel programming. Technical report, The University of Tokyo, Jan. 2011.

[22] H. Wei, H. Tan, X. Liu, and J. Yu. StreamX10: a stream programming framework on X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 1:1–1:6, New York, NY, USA, 2012. ACM.

A Overall structure of the MiX10 compiler

MiX10 is implemented on top of several existing MATLAB compiler tools. The overall structure is given in Figure 3, where the new parts are indicated by the shaded boxes, and future work is indicated by dashed boxes.

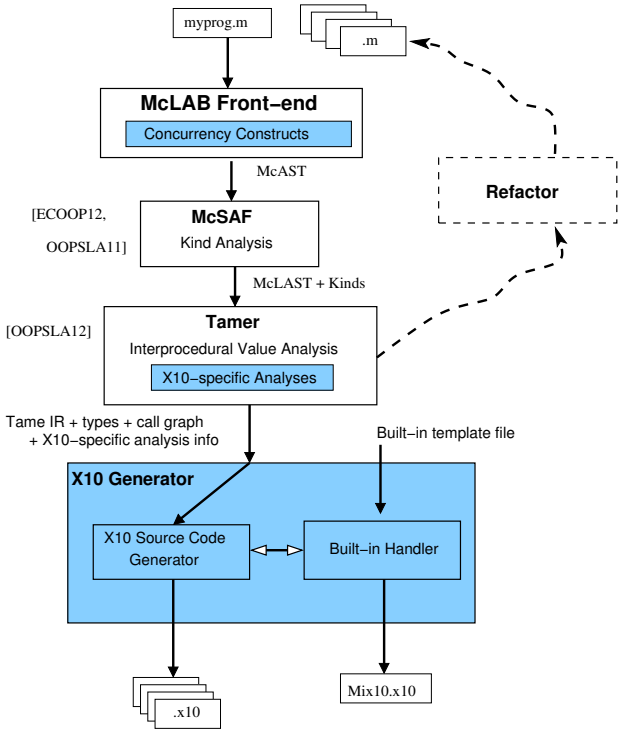

Figure 3: Overview of MiX10 structure

As illustrated at the top of the figure, a MATLAB programmer only needs to provide an entry-point MATLAB function (called `myprog.m` in this example), plus a collection of other MATLAB functions and libraries (directories of functions) which may be called, directly or indirectly, by the entry point. The programmer may also specify the types and/or shapes of the input parameters to the entry-point function. As shown at the bottom of the figure, our MiX10 compiler automatically produces a collection of X10 output files which contain the generated X10 code for all reachable MATLAB functions, plus one X10 file called `mix10.x10` which contains generated and specialized X10 code for the required builtin MATLAB functions. Thus, from the MATLAB programmer's point of view, the MiX10 compiler is quite simple to use.

B Introduction to X10 concurrency controls

X10 is a high performance language that aims at providing productivity to the programmer. To achieve that goal, it provides a simple yet powerful concurrency model that provides four concurrency constructs that abstract away the low-level details of parallel programming from the programmer, without compromising on performance. X10's concurrency model is based on the Asynchronous Partitioned Global Address Space (APGAS) model [8]. The APGAS model has a concept of global address space that allows a task in X10 to refer to any object (local or remote). However, a task may operate only on an object that resides in its partition of the address space (local memory). Each task, called an *activity*, runs asynchronously parallel to each other. A logical processing unit in X10 is called a *place*. Each *place* can run multiple *activities*. The following four types of concurrency constructs are provided by X10 [9]:

B.1 Async

The fundamental concurrency construct in X10 is `async`. The statement `async S` creates a new *activity* to execute `S` and returns immediately. The current activity and the "forked" activity execute asynchronously parallel to each other and have access to the same heap of objects as the current activity. They communicate with each other by reading and writing shared variables. There is no restriction on statement `S` and can contain any other constructs (including `async`). `S` is also permitted to refer to any immutable variable defined in lexically enclosing scope.

An activity is the fundamental unit of execution in X10. It may be thought of as a very light-weight thread of execution. Each activity has its own control stack and may invoke recursive method calls. Unlike Java threads, activities in X10 are unnamed. Activities cannot be aborted or interrupted once they are in flight. They must proceed to completion, either finishing correctly or by throwing an exception. An activity created by `async S` is said to be *locally terminated* if `S` has terminated. It is said to be *globally terminated* if it has terminated locally and all activities spawned by it recursively, have themselves globally terminated.

B.2 Finish

Global termination of an activity can be converted to local termination by using the `finish` construct. This is necessary when the programmer needs to be sure that a statement `S` and all the activities spawned transitively by `S` have terminated before execution of the next statement begins.

B.3 Atomic

`atomic S` ensures that the statement (or set of statements) `S` is executed in a single step with respect to all other activities in the system. When `S` is being executed in one activity all other activities containing `s` are suspended. However, the `atomic` statement `S` must be *sequential*, *non-blocking* and *local*.

Note that, `atomic` is a syntactic sugar for the construct `when (c) .` `when (c)` is the conditional atomic statement based on binary condition (c). Statement `when (c) S` executes statement `S` atomically only when `c` evaluates to true; if it is false, the execution blocks waiting for `c` to be true. Condition `c` must be *sequential*, *non-blocking* and *local*.

B.4 At

A *place* in X10 is the fundamental processing unit. It is a collection of data and activities that operate on that data. A program is run on a fixed number of places. The binding of places to hardware resources (e.g. nodes in a cluster, accelerators) is provided externally by a configuration file, independent of the program.

The `at` construct provides a place-shifting operation, that is used to force execution of a statement or an expression at a particular place. An activity executing `at (p) S` suspends execution at the current place; The object graph G at the current place whose roots are all the variables V used in `S` is serialized, and transmitted to place `p`, deserialized (creating a graph G' isomorphic to G), an environment is created with the variables V bound to the corresponding roots in G', and `S` executed at `p` in this environment. On local termination of `S`, computation resumes after `at (p) S` in the original location. The object graph is not automatically transferred back to the originating place when `S` terminates: any updates made to objects copied by an `at` will not be reflected in the original object graph.