MIX10: A MATLAB TO X10 COMPILER

by

Vineet Kumar

School of Computer Science
McGill University, Montréal

Thursday, October 31st 2013

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

TBD

Résumé

TBD

Acknowledgements

TBD

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

MATLAB is a popular numeric programming language, used by millions of scientists, engineers as well as students worldwide[Mol]. 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.

X10 [IBM12], on the other hand, is an 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. 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.

In this thesis we present MIX10, a source-to-source compiler that helps to bridge the gap between MATLAB, a language familiar to scientists, and X10, a language designed for high performance computing systems. MIX10 statically compiles MATLAB programs to X10 and thus allows scientists and engineers to write programs in MATLAB (or use old programs already written in MATLAB) and still get the benefits of high performance

computing without having to learn a new language.Also, systems that use MATLAB for prototyping and C++ or Java for production, can benefit from MIX10 by quickly converting MATLAB prototypes to C++ or Java programs via X10

On one hand, all the aforementioned characteristics of MATLAB make it a very user-friendly and thus popular application to develop software among a non-programmer community. On the other hand, these same characteristics make MATLAB a difficult language to compile statically. Even the de facto standard, Mathworks' implementation of MATLAB is essentially an interpreter with a *JIT accelarator*[The02] which is generally slower than statically compiled languages. GNU Octave, which is a popular open source alternative to MATLAB and is mostly compatible with MATLAB, is also implemented as an interpreter[Oct]. Lack of formal language specification, unconventional semantics and closed source make it even harder to write a compiler for MATLAB. Furthermore, the use of arrays as default data type and the dynamicity of the base types and shapes of arrays also make it harder to add support for concurrency in a static MATLAB compiler. Mathworks' proprietary solution for concurrency is the *Parallel Computing Toolbox*[Mat13], which allows users to use multicore processors, GPUs and clusters. However, this toolbox uses heavyweight worker threads and has limited scalability.

Built on top of *Mc*LAB static analysis framework[Doh11, DH12], MIX10, together with its set of reusable static analyses for performance optimization and extended support for MATLAB features, ultimately aims to provide MATLAB's ease of use, to benefit from the advantages of static compilation, and to expose scalable concurrency.

1.1 Contributions

The major contributions of this thesis are as follows:

Identifying key challenges: We have identified the key challenges in performing a semantics-preserving efficient translation of MATLAB to X10.

Overall design of MIX10: Building upon the *Mc*LAB frontend and analysis framework, we provide the design of the MIX10 source-to-source translator that includes a low-

level X10 IR and a template-based specialization framework for handling builtin operations.

Static analyses: We provide a set of reusable static analyses for performance optimization and extended support for MATLAB features. These analyses include: (1) *IntegerOkay analysis* - We provide an analysis to automatically identify variables that can be safely declared to be of type `Int` (or `Long`) without affecting the correctness of the generated X10 code. This helps to eliminate most of, otherwise necessary, typecast operations which our experiments showed to be a major performance bottleneck in the generated code; (2) *Variable renaming for type collision* - MATLAB allows a variable to hold values of different types at different points in a program. However, in statically typed languages like X10 this behaviour cannot be supported since a variable's type needs to be declared statically by the programmer and cannot be changed at any point in the program. We provide an analysis to identify and rename such variables if their different types belong to mutually exclusive UD-DU webs; and (3) *isComplex value analysis* - We designed an analysis for identification of complex numerical values in a MATLAB program. This helped us to extend MIX10 compiler to also generate X10 code for MATLAB programs that involve use of complex numerical values.

Code generation strategies for key language constructs: There are some very significant differences between the semantics of MATLAB and X10. A key difference is that MATLAB is dynamically-typed, whereas X10 is statically-typed. Furthermore, the type rules are quite different, which means that the generated X10 code must include the appropriate explicit type conversion rules, so as to match the MATLAB semantics. Other MATLAB features, such as multiple returns from functions, a non-standard semantics for `for` loops, and a very general range operator, must also be handled correctly. MIX10 not only supports all the key sequential constructs but also supports concurrency constructs like `parfor` and can handle vectorized instructions in a concurrent fashion. We have also designed and implemented a template-based system that allows us to generate specialized X10 code for a collection of important MATLAB builtin operations.

Techniques for efficient compilation of 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. 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 and provide techniques to compile MATLAB arrays to two different representations of arrays provided by X10.

Working implementation and performance results: We have implemented the MIX10 compiler over various MATLAB compiler tools provided by the *McLAB* toolkit. In the process we also implemented some enhancements to these existing tools. We provide performance results for different X10 backends over a set of benchmarks and compare them with results from other MATLAB compilers including Mathworks' MATLAB implementation and Octave.

1.2 Thesis Outline

This thesis is divided into 9 chapters, including this one and is structured as follows.

Chapter 2 provides an introduction to the X10 language and describes how it compares to MATLAB from the point of view of language design. *Chapter 3* gives a description of various existing MATLAB compiler tools upon which MIX10 is implemented, presents a high-level design of MIX10, and explains the design and need of MIX10 IR. In *Chapter 6* we provide a description of the *IntegerOkay* analysis to identify variables that are safe to be declared as `Long` type, *variable renaming for type conflict* to rename variables with conflicting types in isolated UD-DU webs and *isComplex* analysis to identify complex numerical values. *Chapter 4* gives details of code generation strategies for important MATLAB constructs. In *Chapter 5* we introduce different types of arrays provided by X10, we identify pros and cons of both kinds of arrays in the context of X10 as a target language and describe code generation strategies for them. *Chapter 7* provides performance results for code generated using MIX10 for a suite of benchmarks. *Chapter 8* provides an overview

of related work and *Chapter 9* concludes and outlines possible future work.

Chapter 2
Introduction to X10 programming language

In this chapter, we describe key X10 semantics and features to help readers unfamiliar with X10 to have a better understanding of the MIX10 compiler.

X10 is an award winning open-source programming language being developed by IBM Research. The goal of the X10 project is to provide a productive and scalable programming model for the new-age high performance computing architectures ranging from multi-core processors to clusters and supercomputers [IBM12].

X10, like Java, is a class-based, strongly-typed, garbage-collected and object-oriented language. It uses Asynchronous Partitioned Global Address Space (APGAS) model to support concurrency and distribution [SBP+13]. The X10 compiler has a *native backend* that compiles X10 programs to C++ and a *managed backend* that compiles X10 programs to Java.

In contrast to X10, MATLAB is a commercially-successful, proprietary programming language that focuses on simplicity of implementing numerical computation application [Mol]. MATLAB is a weakly-typed, dynamic language with unconventional semantics and uses a JIT compiler backend. It provides restricted support for high performance computing via Mathworks' parallel computing toolbox [Mat13].

2.1 Overview of X10's key sequential features

X10's sequential core is a container-based object-oriented language that is very similar

to that of Java or C++ [SBP+13]. A X10 program consists of a collection of classes, structs or interfaces, which are the top-level compilation units. X10's sequential constructs like `if-else` statements, `for` loops, `while` loops, `switch` statements, and exception handling constructs `throw` and `try...catch` are also same as those in Java. X10 provides both, implicit coercions and explicit conversions on types, and both can be defined on user-defined types. The `as` operator is used to perform explicit type conversions; for example, `x as Long{self != 0}` converts `x` to type `Long` and throws a runtime exception if its value is zero. Multi-dimensional arrays in X10 are provided as user-defined abstractions on top of `x10.lang.Rail`, an intrinsic one-dimensional array analogous to one-dimensional arrays in languages like C or Java. Two families of multi-dimensional array abstractions are provided: *simple arrays*, which provide a restricted but efficient implementation, and *region arrays* which provide a flexible and dynamic implementation but are not as efficient as *simple arrays*. Listing 2.1 shows a sequential X10 program that calculates the value of π using the Monte Carlo method. It highlights important sequential and object-oriented features of X10 detailed in the following subsections.

```
package examples;
import x10.util.Random;

public class SeqPi {
  public static def main(args:Rail[String]) {
    val N = Int.parse(args(0));
    var result:Double = 0;
    val rand = new Random();
    for(1..N) {
      val x = rand.nextDouble();
      val y = rand.nextDouble();
      if(x*x + y*y <= 1) result++;
    }
    val pi = 4*result/N;
    Console.OUT.println("The value of pi is " + pi);
  }
}
```

Listing 2.1 Sequential X10 program to calculate value of π using Monte Carlo method

2.1.1 Object-oriented features

A program consists of a collection of *top-level units*, where a unit is either a *class*, a *struct* or an *interface*. A program can contain multiple units, however only one unit can be made `public` and its name must be same as that of the program file. Similar to Java, access to these *top-level units* is controlled by *packages*. Below is a description of the core object-oriented constructs in X10:

Class A class is a basic bundle of data and code. It consists of zero or more *members* namely *fields*, *methods*, *constructors*, and member classes and interfaces [IBM13b]. It also specifies the name of its *superclass*, if any and of the interfaces it *implements*.

Fields A field is a data item that belongs to a class. It can be mutable (specified by the keyword `var`) or immutable (specified by the keyword `val`). The type of a mutable field must be always be specified, however the type of an immutable field may be omitted if it's declaration specifies an *initializer*. Fields are by default instance fields unless marked with the `static` keyword. Instance fields are inherited by subclasses, however subclasses can shadow inherited fields, in which case the value of the shadowed field can be accessed by using the qualifier `super`.

Methods A method is a named piece of code that takes zero or more *parameters* and returns zero or one value. The type of a method is the type of the return value or `void` if it does not return a value. If the return type of a method is not provided by the programmer, X10 infers it as the least upper bound of the types of all expressions `e` in the method where the body of the method contains the statement `return e`. A method may have a type parameter that makes it *type generic*. An optional *method guard* can be used to specify constraints. All methods in a class must have a unique signature which consists of its name and types of its arguments.

Methods may be inherited. Methods defined in the superclass are available in the subclasses, unless overridden by another method with same signature. Method overloading allows programmer to define multiple methods with same name as long as they have different signatures. Methods can be access-controlled to be `private`, `protected` or `public`. `private` methods can only be accessed by other methods in the same class. `protected` methods can be accessed in the same class or its subclasses. `public` methods can be accessed from any code. By default, all methods are *package protected* which means they can be accessed from any code in the same package.

Methods with the same name as that of the containing class are called constructors. They are used to instantiate a class.

Structs A struct is just like a class, except that it does not support inheritance and may not be recursive. This allows structs to be implemented as *header-less* objects, which means that unlike a class, a struct can be represented by only as much memory as is necessary to represent its fields and with its methods compiled to static methods. It does not contain a *header* that contains data to represent meta-information about the object. Current version of X10 (version 2.4) does not support mutability and references to structs, which means that there is no syntax to update the fields of a struct and structs are always passed by value.

Function literals X10 allows definition of functions via literals. A function consists of a parameter list, followed optionally by a return type, followed by =>, followed by the body (an expression). For example, `(i:Int, j:Int) => (i<j ? foo(i) : foo(j))`, is a function that takes parameters `i` and `j` and returns `foo(i)` if `i<j` and `foo(j)` otherwise. A function can access immutable variables defined outside the body.

2.1.2 Statements

X10 provides all the standard statements similar to Java. Assignment, `if - else` and `while` loop statements are identical to those in Java.

`for` loops in X10 are more advanced and apart from the standard C-like for loop, X10 provides three different kinds of for loops:

***enhanced* `for` loops** take an index specifier of the form `i in r`, where `r` is any value that implements `x10.lang.Iterable[T]` for some type `T`. Code listing 2.2 below shows an example of this kind of `for` loops:

```
def sum(a:Rail[Long]):Long{
    var result:Long = 0;
    for (i in a){
        result += i;
    }
    return result;
}
```

Listing 2.2 Example of enhanced for loop

`for` loops over `LongRange` iterate over all the values enumerated by a `LongRange`. A `LongRange` is instantiated by an expression like `e1..e2` and enumerates all the integer values from `a` to `b` (inclusive) where `e1` evaluates to `a` and `e2` evaluates to `b`. Listing 2.3 below shows an example of a `for` loop that uses `LongRange`:

```
def sum(N:Long):Long{
    var result:Long = 0;
    for (i in 0..N){
        result += i;
    }
    return result;
}
```

Listing 2.3 Example of for loop over `LongRange`

`for` loops over `Region` allow to iterate over multiple dimensions simultaneously. A `Region` is a data structure that represents a set of *points* in multiple dimensions. For

instance, a `Region` instantiated by the expression `Region.make(0..5,1..6)` creates a 2-dimensional region of *points* `(x,y)` where `x` ranges over `0..5` and `y` over `1..6`. The natural order of iteration is lexicographic. Listing 2.4 below shows an example that calculates the sum of coordinates of all points in a given rectangle:

```
def sum(M:Long, N:Long):Long{
    var result:Long = 0;
    val R:Region = x10.regionarray.Region.make(0..M,0..N);
    for ([x,y] in R){
        result += x+y;
    }
    return result;
}
```

Listing 2.4 Example of for loop over a 2-D `Region`

2.1.3 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.

2.1.4 Types

X10 is a statically type-checked language: Every variable and expression has a type that is known at compile-time and the compiler checks that the operations performed on an expression are permitted by the type of that expression. The name `C` of a class or an interface is the most basic form of type in X10. There are no primitive types.

X10 also allows *type definitions*, that allow a simple name to be supplied for a complicated type, and for type aliases to be defined. For example, a type definition like `public static type bool(b:Boolean) = Boolean{self=b}` allows the use of expression `bool(true)` as a shorthand for type `Boolean{self=true}`.

Generic types X10's generic types allow classes and interfaces to be declared parameterized by types. They allow the code for a class to be reused unbounded number of times, for different concrete types, in a type-safe fashion. For instance, the listing 2.5 below shows a class `List[T]`, parameterized by type `T`, that can be replaced by a concrete type like `Int` at the time of instantiation (`var l:List[Int] = new List[Int](item)`).

```
class List[T]{
    var item:T;
    var tail:List[T]=null;
```

```
    def this(t:T){
        item=t;
    }
}
```

X10 types are available at runtime, unlike Java(which erases them).

Constrained types X10 allows the programmer to define `Boolean` expressions (restricted) constraints on a type `[T]`. For example, a variable of constrained type `Long{self != 0}` is of type `Long` and has a constraint that it can hold a value only if it is not equal to `0` and throws a runtime error if the constraint is not satisfied. The permitted constraints include the predicates `==` and `!=`. These predicates may be applied to constraint terms. A constraint term is either a final variable visible at the point of definition of the constraint, or the special variable `self` or of the form `t.f` where `f` names a field, and `t` is (recursively) a constraint term.

2.2 Overview of X10's concurrency features

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 [IBM13a]. 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*. Following four types of concurrency constructs are provided by X10 [IBM13b]:

2.2.1 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 activty 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.

2.2.2 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. For instance in the listing 2.5 below, use of `finish` ensures that the `Console.OUT.println("a(1) = " + a(1));` statement is executed only after all the asynchronously executing operations (`async a(i) *= 2;` have completed.

```
//...
//Create a Rail of size 10, with i'th element initialized to i
val a:Rail[Long] = new Rail[Long](10,(i:Long)=>i);
finish for (i in 0..9){
//asynchronously double every value in the Rail
```

```
    async a(i) *= 2;
}
Console.OUT.println("a(1) = " + a(1));
//...
```

Listing 2.5 Example use of `finish` construct

2.2.3 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*. Consider the code fragment in listing 2.6. It asynchronously adds `Long` values to a linked-list `list` and simultaneously holds the size of the list in a variable `size`. The use of `atomic` guarantees that no other operation, in any activity, is executed in between (or simultaneously with) these two operations, which is necessary to ensure correctness of the program.

```
//...
    finish for (i in 0..10){
        async add(i);
    }
//...
    def add(x:Long){
        atomic {
            this.list.add(x);
            this.size = this.size + 1;
        }
    }
//...
```

Listing 2.6 Example use of `atomic` construct

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*.

2.2.4 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.

`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.

2.3 Overview of X10's implementation and runtime

In order to understand the compilation flow of the MIX10 compiler and enhancements made to the X10 compiler for efficient use of X10 as a target language for MATLAB, it is important to understand the design of the X10 compiler and its runtime environment.

2.3.1 X10 implementation

X10 is implemented as a source-to-source compiler that translates X10 programs to either C++ or Java. This allows X10 to achieve critical portability, performance and interoperability objectives. The generated C++ or Java program is, in turn, compiled by the platform C++ compiler to an executable or to class files by a Java compiler. The C++ backend is referred to as *Native* X10 and the Java backend is called *Managed* X10.

The source-to-source compilation approach provides three main advantages: (1) It makes X10 available for a wide range of platforms; (2) It takes advantage of the underlying classical and platform-specific optimizations in C++ or Java compilers, while the X10 implementation includes only X10 specific optimizations; and (3) It allows programmers to take advantage of the existing C++ and Java libraries.

Figure 2.1 shows the overall architecture of the X10 compiler [IBM13b].

2.3.2 X10 runtime

Figure 2.2 shows the major components of the X10 runtime and their relative hierarchy [IBM13b].

The runtime bridges the gap between application program and the low-level network transport system and the operating system. X10RT, which is the lowest layer of the X10 runtime, provides abstraction and unification of the functionalities provided by various network layers.

The X10 Language Native Runtime provides implementation of the sequential core of the language. It is implemented in C++ for native X10 and Java for Managed X10.

XRX Runtime, the X10 runtime in X10 is the core of the X10 runtime system. It provides implementation for the primitive X10 constructs for concurrency and distribution (`async`, `finish`, `atomic` and `at`). It is primarily written in X10 over a set of low-level APIs that provide a platform-independent view of processes, threads, synchronization mechanisms and inter-process communication.

At the top of the X10 runtime system, is a set of core class libraries that provide fundamental data types, basic collections, and key APIs for concurrency and distribution.

2.4 Summary

In this chapter we have provided an overview of the key features of the X10 programming language. In the following chapters, specially chapters *Chapter 4* and *Chapter 5*, we will discuss some of the features and constructs introduced here, in more depth.

We will also discuss key constructs and features of the MATLAB programming language

and contrast them with X10, as we discuss MIX10's compilation strategies in the following chapters. For readers who are completely unfamiliar with MATLAB or are interested in a quick overview, we suggest reading chapter 2 of [Dub12].

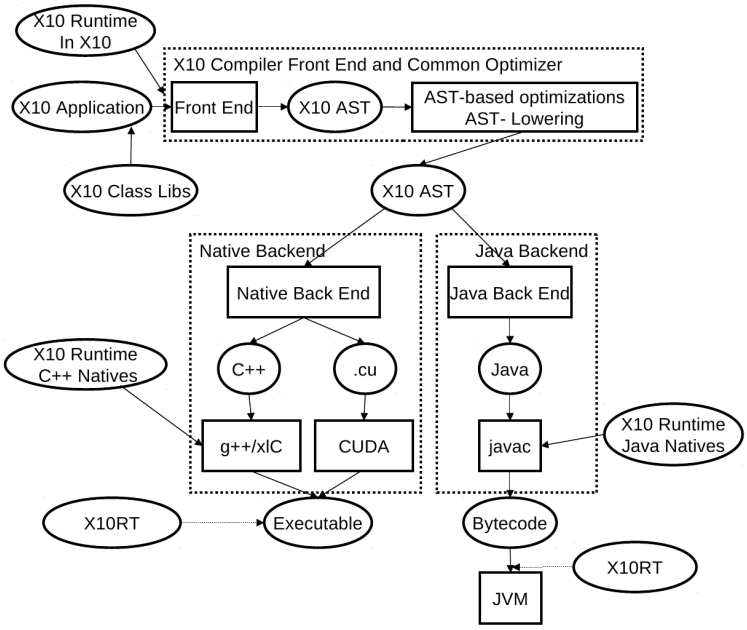

Figure 2.1 Architecture of the X10 compiler

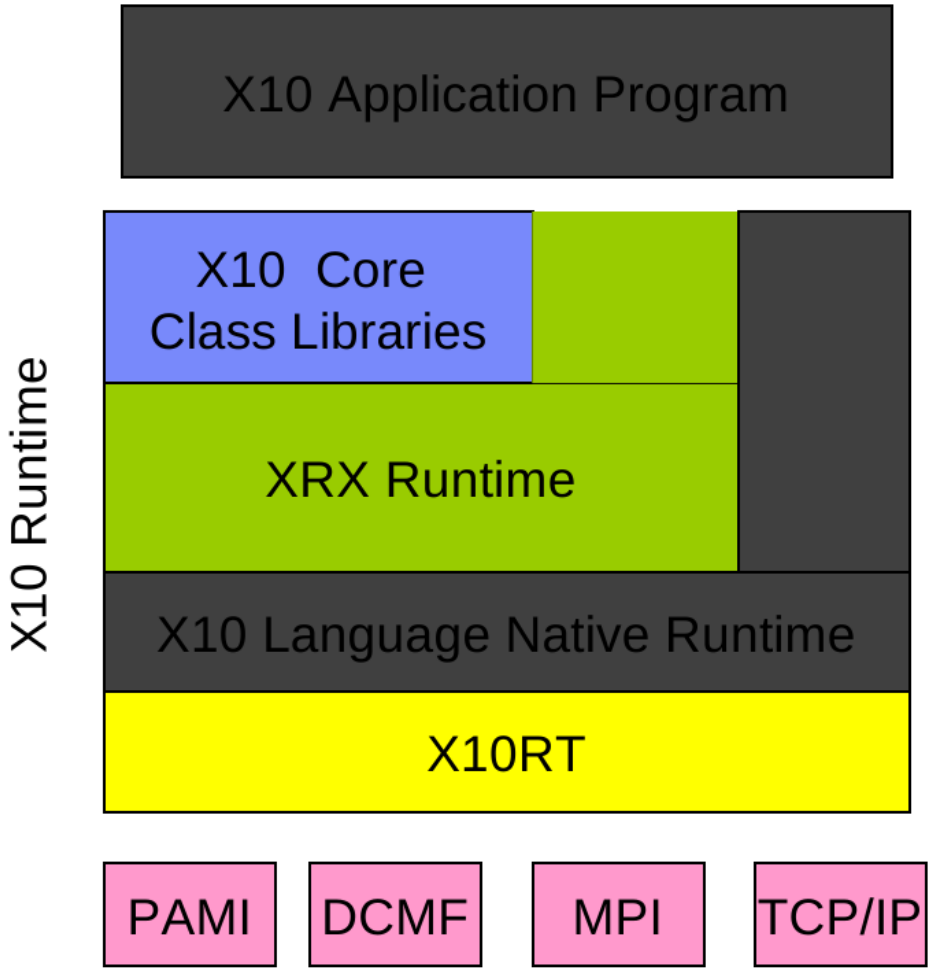

Figure 2.2 Architecture of the X10 runtime

Chapter 3
Background and High level design

TBD

Chapter 4
Code generation

TBD

Chapter 5
Techniques for efficient compilation of MATLAB arrays

TBD

Chapter 6
Static analyses for performance and extended feature support

Chapter 7
Evaluation

In this chapter we present the results of our experiments that we performed to evaluate the correctness and performance of our compiler. We demonstrate the effects on performance due to the differnt approaches to compile arrays, and due to togglingi of various compilation switches. We also compare our results to those of other MATLAB compilers including the de facto Mathworks' compiler, MATLAB coder (also provided by Mathworks) [], and the MC2FOR [] compiler.

The set of benchmarks used for our experiments consists of benchmarks from various sources; Most of them are from related projects like FALCON [] and OTTER [], Chalmers university of Technology [], "Mathworks' central file excahnge" [], and the presentation on parallel programming in MATLAB by Burkardt and Cliff [].

Below is a brief description of each benchmark that we used:

- *bbai* is an implementation of the Babai estimation algorithm and consists of random number generation and operations on a 2-dimensional array in a loop.
- *bubl* is the standard bubble sort algorithm. It is characterized by nested loops and read/write operations on a large row vector.
- *capr* computes the capacitance of a transmission line using finite difference and Gauss-Seidel method. It involves loop-based operations on two 2-dimensional arrays.

- *clos* calculates the transitive closure of a directed graph. Its key feature is matrix multiplication operation on two large 2-dimensional arrays.
- *crni* computes the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a very large (2300×2300) array.
- *dich* computes Dirichlet solution to Laplace's equation. It involves loop-based operation on a 2-dimensional array.
- *diff* calculates diffraction pattern of monochromatic light. Its key feature is explicit array growth via concatenation operation.
- *edit* calculates the edit distance between two strings. It involves operations on large row vectors of `chars`.
- *fiff* computes the finite difference solution to the wave equation. It also involves loop-based operations on a 2-dimensional array.
- *lgdr* calculates derivatives of Legendre polynomials. It is characterized by transpose operation on row vectors.
- *mbrt* computes Mandelbrot sets. It involves operations on scalar data of `complex` type. It also involves `parfor` loop.
- *nb1d* simulates the 1-dimensional n-body problem. It involves operations on column vectors in nested loops including a `parfor` loop.
- *matmul* implements the naive matrix multiplication. It involves three nested loops including one `parfor` loop, and read/write operations on three large 2-dimensional arrays.
- *mcpi* calculates the value of π using the Monte carlo algorithm. It involves random number generation in a loop. It also uses `parfor` loop.
- *numprime* calculates the number of prime numbers upto a given value using the sieve of eratosthenes. It features a `parfor` loop and simple scalar operations.

- *optstop* solves the optimal stopping problem. It involves operations on a row vector, random number generation and a `parfor` loop.
- *quadrature* simulates the quadrature approach to calculate integral of a function. It involves scalar values and a `parfor` loop.

We used 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 MATLAB runtimes (normalized to one). We summarize our results in the following sections:

7.1 MIX10 performance comparison with MATLAB, MATLAB coder, MC2FOR, and Octave

We compared the performance of the generated X10 code with that of the original MATLAB code run on Mathworks' implementation of MATLAB, and Octave. We also compared it to the generated C code and Fortran code by MATLAB coder and MC2FOR compilers respectively.

Figure 7.1 and *Figure 7.2* show the speedups and slowdowns for codes generated for our benchmarks by different compilers. For MIX10 we have included results for generated X10 code by the *x10c++* compiler with 1) Array bounds checks turned off; and 2) Array

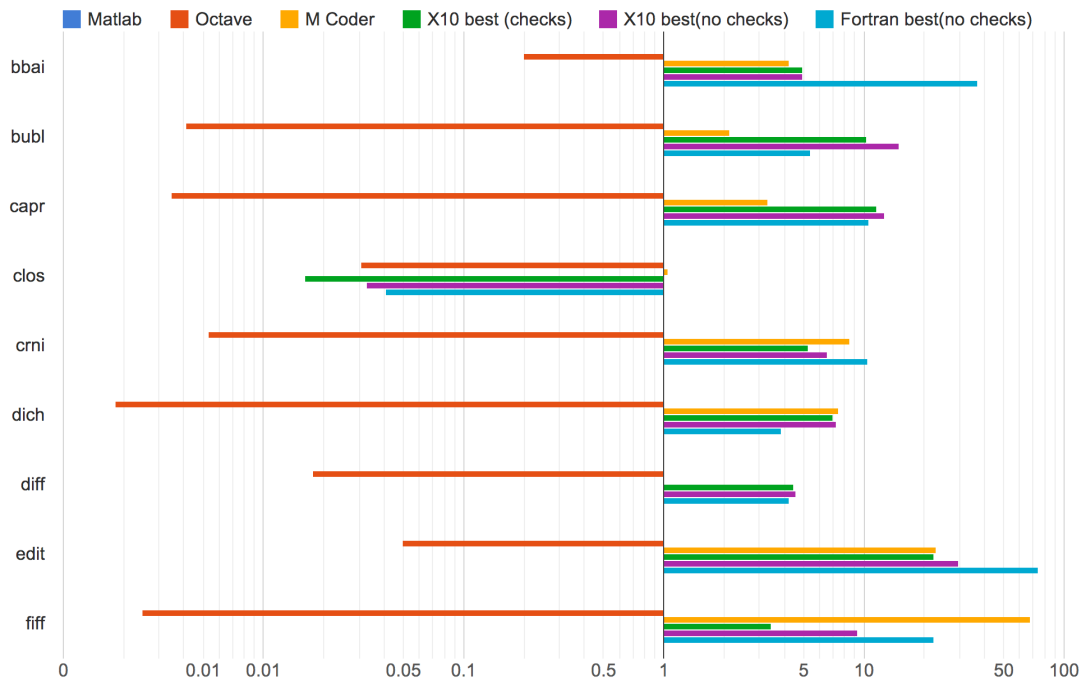

Figure 7.1 MiX10 performance comparison (part1)

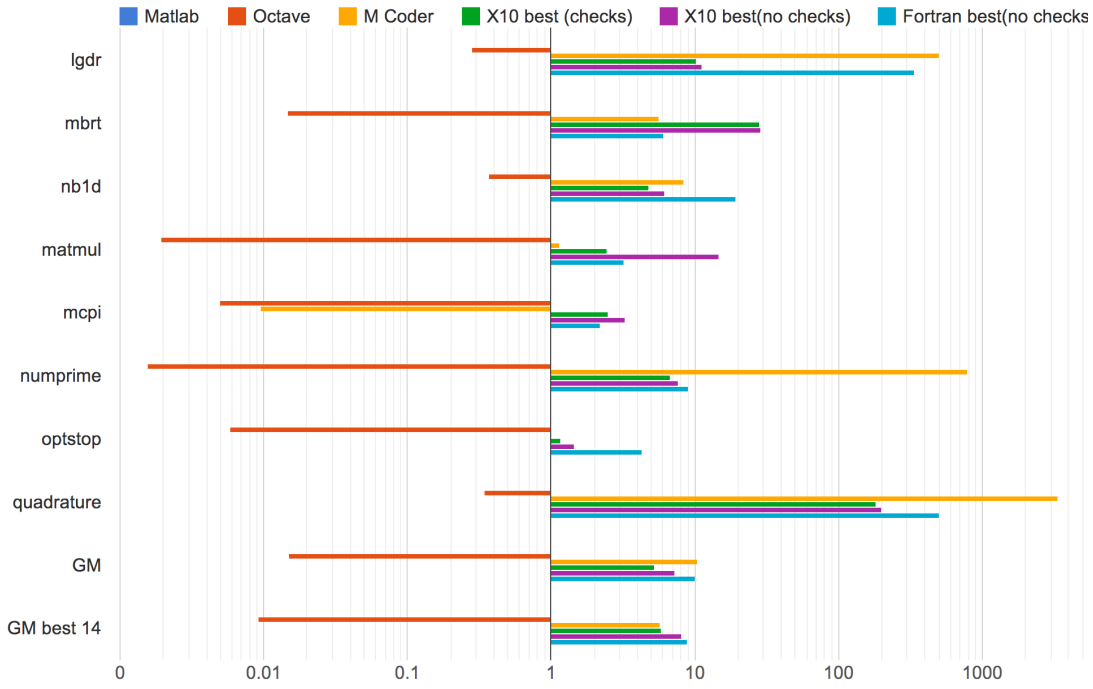

Figure 7.2 MiX10 performance comparison (part2)

bounds checks turned on. We used the default optimization provided by the X10 compiler (`-O`). For Fortran we included the code generated without bounds checks. C code from MATLAB coder was generated with default settings and includes bounds checks. *Figure 7.2* also shows the geometric mean of speedups for all the benchmarks and for our best 14 out of 17 results (compared to results from MATLAB coder).

We achieved a mean speedup of 5.2 and 7.2 for x10c++ with bounds checks and x10c++ with no bounds checks respectively. On the other hand MATLAB coder gave a mean speedup of 10.5 and MC2FOR gave a mean speedup of 10.2. However, we see that if we do not consider only three benchmarks for which X10 does not perform as well as C and Fortran, we get a mean sppedup of 5.75 for x10 with bounds checks compared to 5.72 for C. For no bounds checks we get a mean speedup of 8.1 compared to 8.8 for Fortran. We outperform MATLAB coder in 8 out of 17 benchmarks, and Fortran in 7 out of 17 benchmarks

clos involves builtin matrix multiplication operations for 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, we get a speedup of 0.016 as compared to 1.049 for C. Note that the generated Fortran code is also slowed down (speedup of 0.041) due to the same reason.

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 as highly optimized. However, we still achieved a speedup of over 10 times.

quadrature solves the stadard quadrature formula for numerical integration [], which 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.

Other interesting numbers are shown by *optstop*, *numprime* and *fiff*. *optstop* involves repeated array indexing by an index of type `Double` which needs to be explicitly cast to `Long`, whenever used as an index. IntegerOkay analysis cannot convert it to an integer type because it's the value returned from a call to the `floor()` function whose return

type is `Double`. *numprime* involves similar problem, where the result of the `sqrt()` function needs to be converted to `Long` from `Double`. *numprime* also involves a for loop over a conditional that evaluates to true only once. MATLAB coder leverages this fact for optimizing the for loop by implicitly inserting a `break` when the conditional becomes true. We tested our generated X10 code by explicitly inserting a `break` statement and achieved a speedup of around 65 times. Note that *numprime* is also used for demonstrating `parfor` which does not allow a `break` statement in the loop. *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. Note that 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.

For most of the other benchmarks, we perform better or nearly equal to C and Fortran code. Inspite of the facts that 1) The sequential core of X10 is a very 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 and Fortran.

7.2 Comparison of X10 C++ backend and X10 Java backend

In this section we present the performance comparison for the MIX10 generated X10 code compiled with X10's C++ backend and that compiled with the X10's Java backend. *Figure 7.3* shows speedups(slowdowns) for our benchmarks compiled with 1) X10 C++ backend without bounds checks, 2) X10 C++ backend with bounds checks, 3) X10 Java backend without bounds checks, and 4) X10 Java backend with bounds checks. We also show the geometric mean for all our benchmarks.

The mean speedups for the C++ backend are 7.22 and 5.16 respectively for bounds checks switched off, and bounds checks switched on, whereas for Java backend these values are 3.99 and 2.81 respectively. This is expected, given that C++ is compiled to

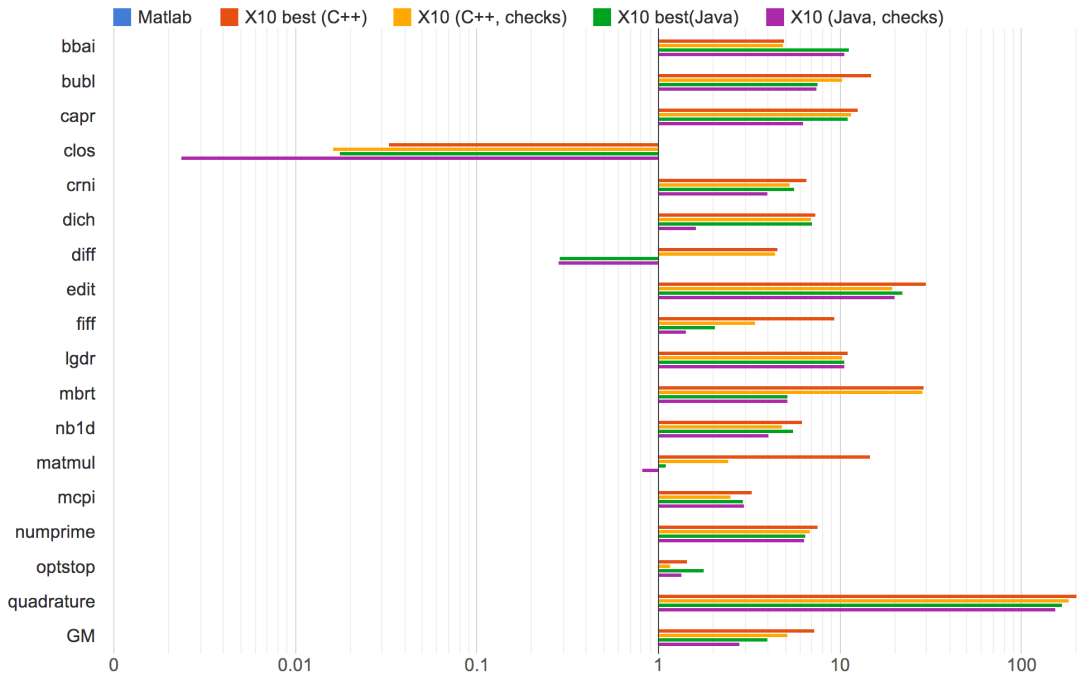

Figure 7.3 MiX10 performance comparison : X10 C++ backend vs X10 Java backend

native binary while Java is JIT compiled. *bbai* and *optstop* are the two noticeable exceptions. *bbai* is slow 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 two reasons :

1. As described before, it uses a value of type `Double` as an array index, thus requiring a typecast to `Long`. This is a costly operation since it involves a check on the value of `Double` type variable to make sure it lies within the range of `Long` type.

2. It also involves random number generation.

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. *diff* also involves explicit array resizing via MATLAB's concatenation operations which we found to be slower in the Java backend. *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(2.06 vs. 9.31 for *fiff* and 1.11 vs. 14.72 for *matmul*), however compared to the performance by the C++ beckend with bounds checks, it is not as slow(2.06 vs. 3.43 for *fiff* and 1.11 vs. 2.45 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.

Another important thing to note is that for the benchmarks *capr* and *dich*, in the case when X10 bounds checks are switched on, we had to turn off the X10 compiler's optimization switch(`-O`) to get useful performance. With the X10 compiler's optimization switch turned on we got a slowdown of over 2 orders of magnitude. We recorded running times of 970 seconds for *capr* compared to 3.2 seconds without the optimization, and 1856 seconds

for *dich* compared to 12.7 seconds without the optimization. With the help of the X10 development team we figured out that switching on the optimization triggered code inlining that caused the resultant Java program to be too big to be handled by the JIT compiler and reverted to interpretation.

7.3 Comparison of Simple arrays, Region Arrays and specialized Region arrays

In this section we discuss the effects on performance due to the use of different kinds of arrays as described in *Chapter 5*. *Figure 7.4* shows the relative speedups and slowdowns for our benchmarks compiled to use different kinds of X10 arrays for the C++ backend. *Figure 7.5* shows the same results for the Java backend. For this experiment we considered all the results with X10 compiler optimizations turned on, and bounds checks turned off.

For the C++ backend we obtained a mean speedup of 7.22 for Simple arrays, compared to 2.94 for region arrays and 3.02 for specialized region arrays. For the Java backend, we obtained speedups of 3.99, 1.25 and 1.86 for the simple arrays, region arrays, and specialized region arrays respectively. These results are as expected in *Chapter 5*. 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. Performance difference is most noticeable for *nb1d*, where region arrays are about 20 times slower than the simple arrays. This is because *nb1d* involves simple operations on a large 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 region arrays and specialized region arrays.

For the Java backend we obtained higher difference in the mean performance for the

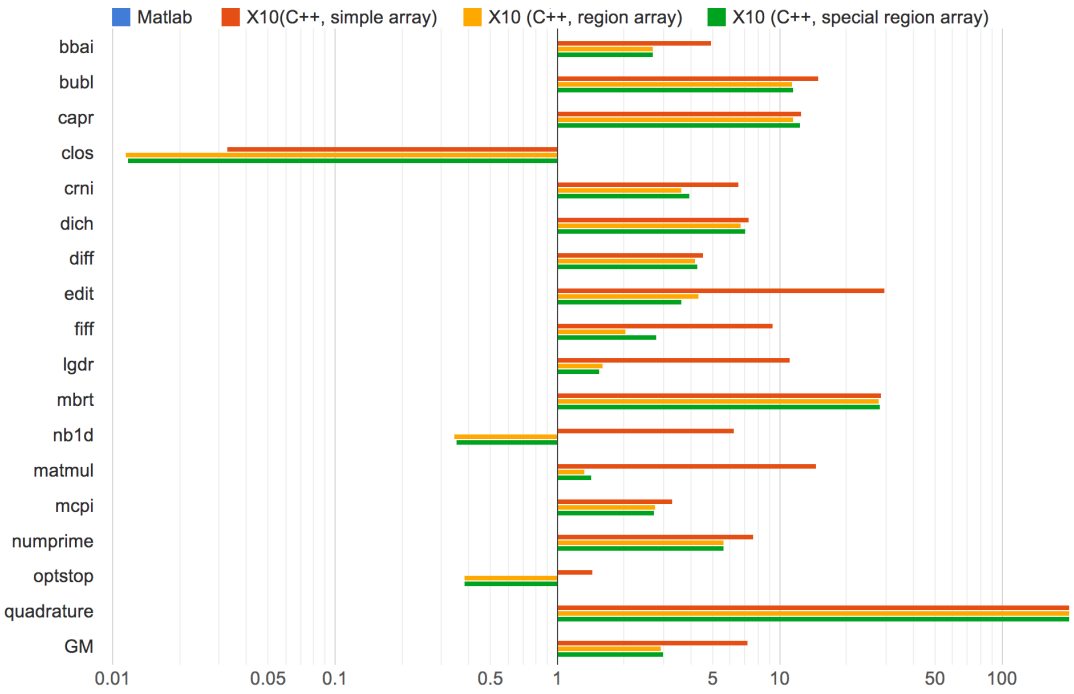

Figure 7.4 MiX10 performance comparison : Simple arrays vs. Region arrays vs. specialized Region Arrays with X10 C++ backend

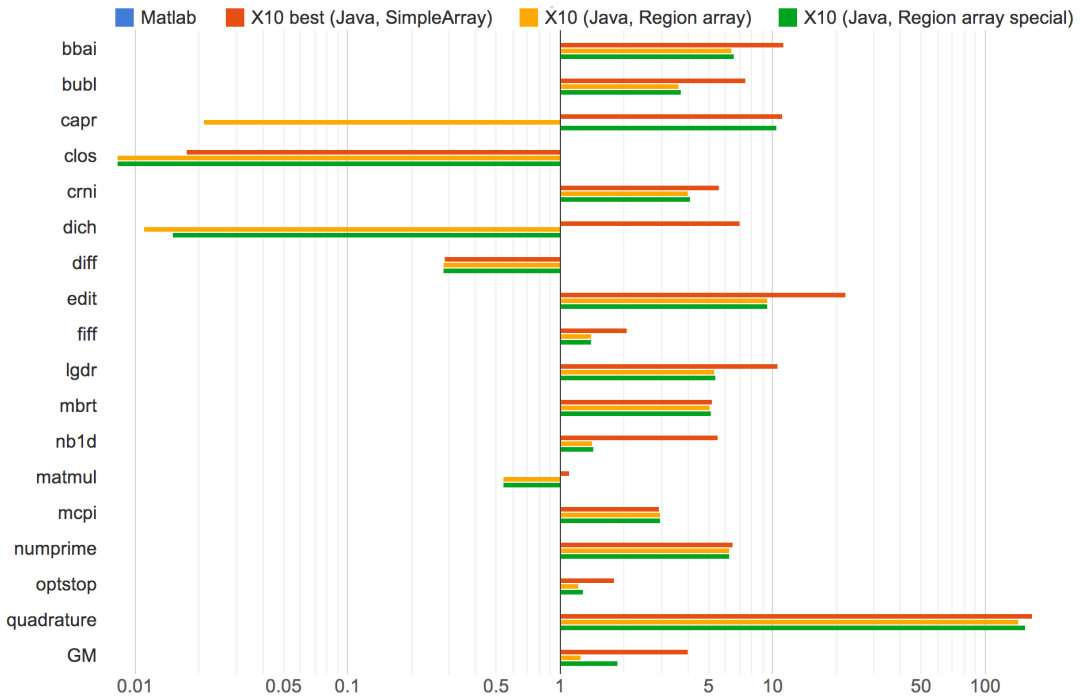

Figure 7.5 MiX10 performance comparison : Simple arrays vs. Region arrays vs. specialized Region Arrays with X10 Java backend

simple arrays and the region arrays. The mean speedup for region arrays is 1.25 whereas for simple arrays it is over 3 times more at 3.99. There is also a significant difference between the performance of specialized region arrays and region arrays. Speedup for specialized region arrays is 1.86. Like the C++ backend, here also, most noticeable performance gain for simple arrays id 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.

7.4 Performance improvements due to the IntegerOkay analysis

In this section we present an overview of the performance improvements achieved by MIX10 by using the IntegerOkay analysis. *Figure* **??** and *Figure* **??** show 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 7.22 times which is more than two times the speedup gained by not using the IntegerOkay analysis, which is equal to 3.44. We observed a significant gain in performance by using IntegerOkay analysis for benchmarks that involve significant number of array indexing operations. *bubl*, *capr*, *crni*, *dich*, *fiff* and *mbrt* 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 require to 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.

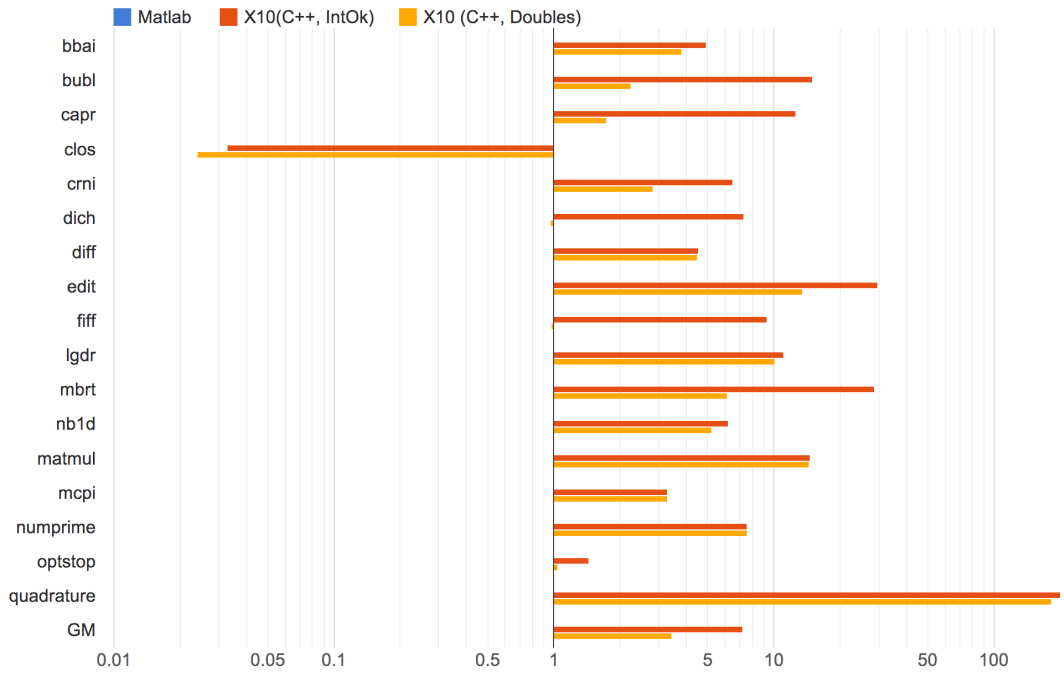

Figure 7.6 MiX10 performance comparison : IntegerOkay analysis vs. all `Double`s with the X10 C++ backend

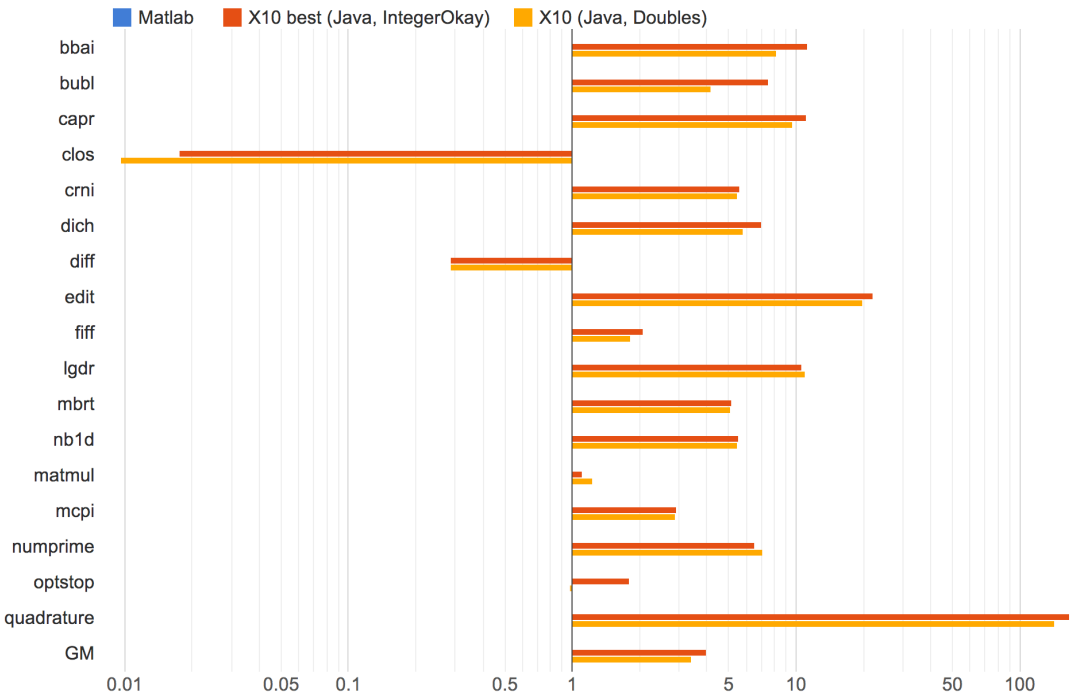

Figure 7.7 MiX10 performance comparison : IntegerOkay analysis vs. all `Double`s with the X10 Java backend

For the Java backend, with the IntegerOkay analysis, we get a mean speedup of 3.99 as compared to 3.41 without it. The reason for lower difference 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 since these checks are performed by the JVM which makes them less time consuming. However, we still notice significant performance difference for the benchmarks which involves read/write operations on arrays of random numbers. For example, *optstop*, and *bubl* both are nearly twice as fast with the IntegerOkay analysis as without it.

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

7.5 performance comparison for MATLAB parfor with the MIX10 generated parallel code

In this section 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 code for these benchmarks to that of MATLAB code using `parfor` and to their sequential X10 version. We used the sequential version with optimizations and no bounds checks and parallel version with optimizations and bounds checks, and with optimizations and without bounds checks. *Figure 7.8* and *Figure 7.9* show the results for the X10 C++ and the X10 Java backends respectively.

For the X10 C++ backend we achieved a mean speedup of 19.82 for the genereated 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 10.99, it is almost twice as fast. Even with the parallel X10 code with bounds checks we achieved a speedup of 12.07 which is 5 times better than the MATLAB `parfor` code. *optstop* is an interesting exception. It is actually slower (at a speedup of 2.12) than the MATLAB `parfor` version (at a speedup of 4.08). The sequential version of *optstop* is just slightly faster than the sequential MATLAB version, with a speedup of

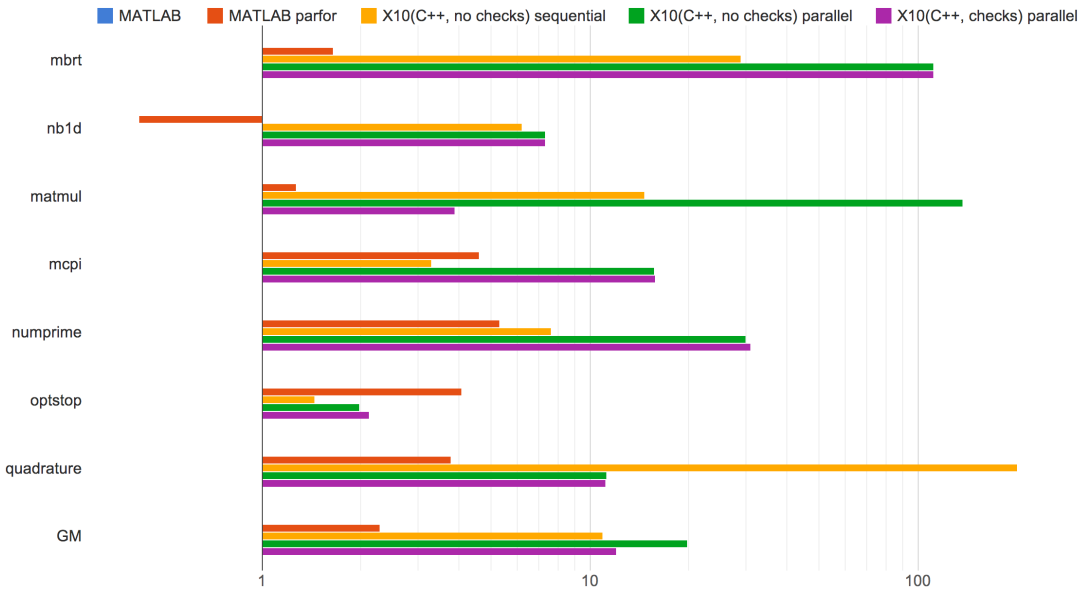

Figure 7.8 MiX10 performance comparison : MATLAB `parfor` vs. MiX10 generated parallel X10 code with X10 C++ backend

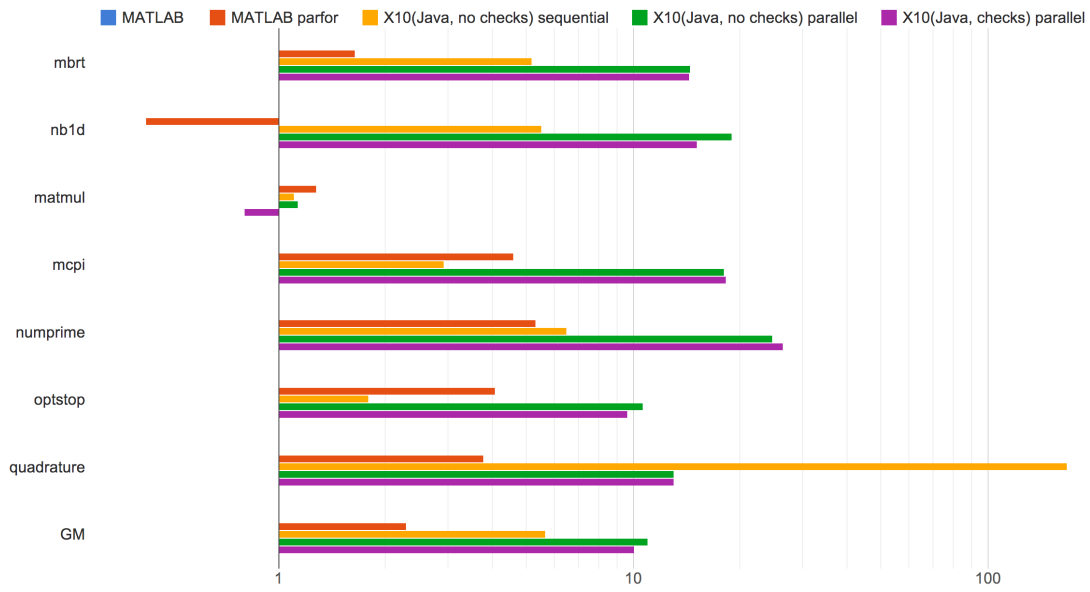

Figure 7.9 MiX10 performance comparison : MATLAB `parfor` vs. MIX10 generated parallel X10 code with X10 Java backend

just 1.45 (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 on) as compared to 1.28 for the MATLAB parfor version. Overall, for the Java backend we obtained a mean speedup of 11.00 for the X10 code with no bounds checks and 10.09 for the code with bounds checks. Compared to the MATLAB parfor version we obtained over 4 times better performance. The parallel X10 code is almost 2 times faster than the sequential X10 code (speedup of 5.65).For both, the C++ backend and the Java backend, the mean speedup for the sequential X10 code is 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.

7.6 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 native code generated by 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 it further in future.

Chapter 8
Related work

Chapter 9
Conclusions and Future Work

TBD

Appendix A
XML structure for builtin framework

TBD

Appendix B
isComplex analysis Propagation Language

TBD

Appendix C
MIX10 IR Grammar

TBD

Bibliography

[DH12] Anton Dubrau and Laurie Hendren. Taming MATLAB. In *Proceedings of OOPSLA 2012*, 2012, pages 503–522.

[Doh11] Jesse Doherty. McSAF: An Extensible Static Analysis Framework for the MATLAB Language. Master's thesis, McGill University, December 2011.

[Dub12] Anton Dubrau. Taming MATLAB. Master's thesis, McGill University, April 2012.

[IBM12] IBM. X10 programming language. http://x10-lang.org, February 2012.

[IBM13a] IBM. Apgas programming in x10 2.4, 2013. http://x10-lang.org/documentation/tutorials/apgas-programming-in-x10-24.html.

[IBM13b] IBM. An introduction to x10, 2013. http://x10.sourceforge.net/documentation/intro/latest/html/intro-web.html.

[Mat13] MathWorks. Parallel computing toolbox, 2013. http://www.mathworks.com/products/parallel-computing/.

[Mol] Cleve Moler. The Growth of MATLAB and The MathWorks over Two Decades. http://www.mathworks.com/company/newsletters/news_notes/clevescorner/jan06.pdf.

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

[SBP+13] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification, Version 2.4*. December 2013.

[The02] The Mathworks. Technology Backgrounder: Accelerating MATLAB, September 2002. http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf.