Compiling X10 to Java

Mikio Takeuchi Yuki Makino[†] Kiyokuni Kawachiya Hiroshi Horii
Toyotaro Suzumura Toshio Suganuma Tamiya Onodera

IBM Research - Tokyo [†]IBM Yamato Software Development Laboratory
1623-14, Shimo-tsuruma, Yamato, Kanagawa 242-8502, Japan
{mtake, makino, kawatiya, horii, toyo, suganuma, tonodera}@jp.ibm.com

Abstract

X10 is a new programming language for improving the software productivity in the multicore era by making parallel/distributed programming easier. X10 programs are compiled into C++ or Java source code, but X10 supports various features not supported directly in Java. To implement them efficiently in Java, new compilation techniques are needed.

This paper discusses problems in translating X10-specific functions to Java and provides our solutions. By using appropriate implementations, sequential execution performance has been improved by about 5 times making it comparable to native Java. The parallel execution performance has also been improved and the gap from Java Fork/Join performance is about 3 times when run at a single place. Initial evaluation of distributed execution shows good scalability. Most of the results in this paper have already been incorporated in X10 release 2.1.2.

Many of the compilation techniques described in this paper can be useful for implementing other programming languages targeted for Java or other managed environments.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, compilers, optimization

General Terms Languages, Design, Performance, Experimentation

Keywords X10, Java, code generation, optimization, evaluation

1. Introduction

For long years, computers have been accelerated through the increase of clock or pipeline speeds. However, such improvements inside a single core reached their limits, and modern processors contain multiple computing cores. By interconnecting such multicore processors through a high-speed network, large-scale parallel-distributed environments are becoming popular. In such environments, the number of total processing cores can be 1,000's to 100,000's, and it is not easy to develop software that can fully utilize them. For that purpose, programming environments also need to evolve for the "multicore era". X10 is a new programming language for improving software productivity in the multicore era by making parallel/distributed programming easier [3]. X10 is now being developed as an open source project led by IBM Research [21].

Currently, X10 programs are compiled into C++ or Java environments. However, X10 supports various features not supported directly in Java, such as new data types and distributed execution. To implement them efficiently in Java, new compilation techniques are needed. For example, in Java, a generic type was introduced after the initial design of the language, and the type parameters are erased by the Java compiler after the type check (type erasure) [1]. However, in X10, generic typing is a core part of its type system. Classes that have different type parameters are treated as different classes (type reification). In addition, X10 can treat not only objects but also structs and functions as first-class data. In contrast to Java, arrays are not one of the language constructs in X10, but are provided as a generalized class. In addition, X10 provides parallel and distributed processing as language-level functions. For example, a parallel "activity", which is a kind of lightweight thread, can be created by the `async` statement. This activity can then move by using an `at` statement to another "place", which roughly corresponds to another node in a distributed environment.

This paper discusses difficulties in translating such X10-specific functions to Java and provides our solutions. These were necessary to satisfy the X10 language specification, but naive implementations may degrade the execution performance. This paper also describes efficient implementations of these X10 features in Java. By using appropriate implementations, sequential execution performance has been improved by about 5 times and it is now comparable to Java. The parallel execution performance has also been improved and the gap from Java Fork/Join performance is about 3 times when run at a single place. Initial evaluation of distributed execution shows good scalability. Most of the results in this paper have already been incorporated in X10 release 2.1.2.

2. X10 Overview

X10 is a statically typed object-oriented language like Java, but has advanced features that are not supported directly in Java, such as new data types like structs and functions with powerful generics and native support for parallel/distributed computing. Therefore, it is not straightforward to compile X10 programs to Java.

In this section, we briefly describe the X10 specification by focusing on its differences from Java that are key for understanding this paper. For a more detailed specification, please refer to the language specification [18].

2.1 Sequential Core

Figure 1 is a sample program to show the basic functions of X10. X10 is a statically typed object-oriented language, and its sequential code resembles Java. Differences are that variables and fields are declared using `val` or `var`, and methods are declared with `def`. Unlike Java, operators can also be declared using `operator`. A type is specified after an entity using a colon, but can be omitted if it is inferable.

As in Java, an X10 program is executed from the `main` method (line 10) of the specified class. X10 supports generics in which the type parameters are specified using "`[ ]`" (line 1). In X10, type parameters are not erased during the compilation, therefore program can introspect the information during execution (line 16). However, X10 does not support dynamic class loading in favor of optimization opportunity by whole program analysis.

In addition to classes and interfaces, X10 supports *structs* and *functions* as first-class data types. A struct represents a small set of immutable data, and is passed by value. It is defined by `struct` keyword (line 6), and cannot be extended. Its size can be determined at the time of compilation. A struct does not have a ref-

X10'11, June 4, 2011, San Jose, California, USA.

```
1  class Sample[T] implements (String)=>String {
2    var data:T;
3    def this(d:T) { data = d; } // constructor
4    public operator this(str:String) = str + data;
5
6    static struct MyPair[T,U](fst:T,snd:U) {
7      public def toString() = "(" + fst + "," + snd +")";
8    }
9
10   public static def main(args:Array[String](1)) {
11     /* Class example */
12     val o = new Sample[Double](1.2);
13     Console.OUT.println(o.data);          // -> 1.2
14     Console.OUT.println(o("Data is ")); // -> Data is 1.2
15     var a:Any = o;
16     Console.OUT.println(a.typeName());  // -> Sample[x10.lang.Double]
17     /* Struct example */
18     val p = MyPair[Int,Double](1,2.3);
19     Console.OUT.println(p);               // -> (1,2.3)
20     val x = 4;
21     /* Function example */
22     val q = MyPair[(Int)=>Int, Int]((i:Int)=>i*x, 5);
23     Console.OUT.println(q.fst(q.snd));  // -> 20
24     /* Array example */
25     val pt = [2,4] as Point;              // Point{rank==2}
26     val R1 = (1..2)*(3..5);               // Region{rank==2}
27     val arr = new Array[Int](R1);         // Array[Int]{rank==2}
28     arr(2,4) = 8;
29     Console.OUT.println(arr(pt));         // -> 8
30     /* Parallel processing */
31     var m:Int = 0; val i = 1;             // mutable/immutable data
32     finish async { m = i * 2; }
33     Console.OUT.println(m);               // -> 2
34     /* Distributed processing */
35     at (here.next()) o.data = 3.4;       // copy of o is modified
36     Console.OUT.println(o.data);          // -> 1.2
37     /* GlobalRef example */
38     val g = GlobalRef(o);
39     at (here.next()) { at (g.home) g().data = 5.6; }
40     Console.OUT.println(o.data);          // -> 5.6
41   }
42 }
```

Figure 1. A Sample X10 Program

erence, and = and == denote substitution and comparison of the contained values, respectively. In X10, primitive types such as `x10.lang.Int` and `x10.lang.Double` are also defined as structs. A function is similar to closure or lambda-expression in other languages, and can be invoked with specified arguments to return a value. X10 supports functions as first-class data, therefore they can be stored into variables or passed as arguments (line 22). Functions can be invoked in a similar manner as methods by specifying arguments in parentheses (line 23). Function types and data are declared using "=>" (line 22). As interfaces, functions can be implemented by classes or structs (line 1). When such a class or struct is used as a function, an "`operator ()`" (line 4) that matches the argument types of the function is executed (line 14).

X10 does not have built-in arrays like Java, but provides a class named `x10.array.Array[T]`, which represents an array of type `T` values. The `Array` is very flexible and can also express multi-dimensional or sparse arrays. The `Array` is indexed by a class `x10.array.Point`, which represents a point on the n-dimensional integer lattice (line 25). A set of `Points` can be represented by a class `x10.array.Region` (line 26), which can be used to specify valid indices (i.e. domain) of an `Array` (line 27). Access to an `Array` data is done by specifying the indices in parentheses after its name (line 29). For `Arrays` whose dimensions are less than 5, `Int` values can be also used as indices (line 28). Internally, "`operator ()=`" and "`operator ()`" are executed at array accesses. By defining these operators, users can also provide their own arrays (such as associative arrays).

Figure 2 illustrates the hierarchical relationship of classes, structs, functions, and interfaces in X10. Solid arrows indicate inheritance (declared by `extends`), and dashed lines indicate implementations (declared by `implements`). For classes, X10 has a similar structure to Java, which is a tree rooted by the class `x10.lang.Object`. In X10, the interface `x10.lang.Any` exists above the `Object`. As shown in the figure, all structs and functions are direct children of `Any`, and there is no hierarchical relationship

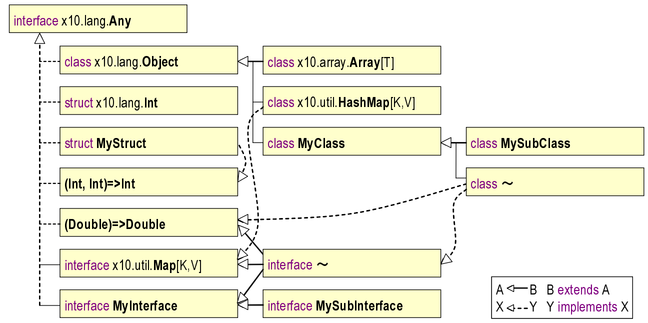


Figure 2. Type Hierarchy in X10

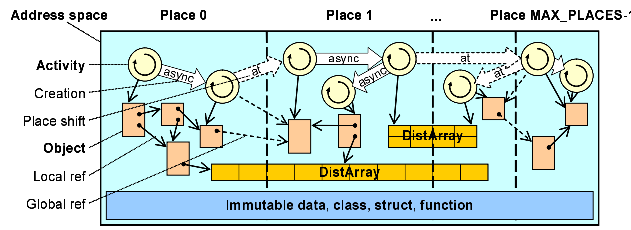


Figure 3. APGAS Programming Model

among them. However, classes and structs can implement multiple interfaces and functions. Since all X10 types implicitly implement `Any`, all X10 data can be stored into a variable of type `Any` (line 15). In X10 source code, the package names `x10.lang` and `x10.array` can be omitted.

2.2 Parallel/Distributed Processing

Figure 3 illustrates the programming model of X10. For parallel/distributed programming, it is very important that the parallelism and memory configuration be exposed to the programmers. X10 uses APGAS (Asynchronous Partitioned Global Address Space) [15] as its programming model, where a global address space is partitioned into multiple *places*. Place is an abstraction of a process that runs on a node. In each place, multiple *activities* and *objects* can exist.

An activity is a lightweight execution thread, running asynchronously in a place. It resembles threads in Java, but the granularity is much smaller, therefore an efficient implementation is required. An activity can be dynamically created in the same place by using an `async` statement (line 32). The activity can even access local variables declared outside of the `async` block. The `finish` statement is used to wait for the termination of activities created inside the block. An activity can move to another place with an `at` statement (line 35).

The object is a mutable data structure and belongs to a specific place. To access an object, activities should be at the same place. If an object is accessed from another place, a copy of the object is implicitly created (lines 35–36). To suppress the implicit copy, an object can be put into an `x10.lang.GlobalRef`, which provides a global reference (line 38). The `GlobalRef` contains place information where the object exists, therefore the activity can access the object by moving to the place (line 39).

In X10, mutable data belongs to a specific place and can be accessed only by the activities in the same place. In contrast, immutable data can be accessed (read) from any place. Examples are classes, structs, and functions. Since the class is an immutable data structure in X10, all static fields must be declared as `val`. Static initializers are executed in `x10.lang.Place.FIRST_PLACE`, where the program is started, and static fields are copied to other places after their initializations. Details of static initialization is described in Section 7.2.

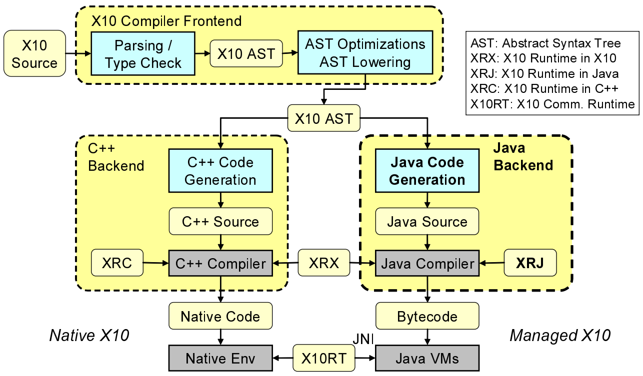


Figure 4. Compilation Flow

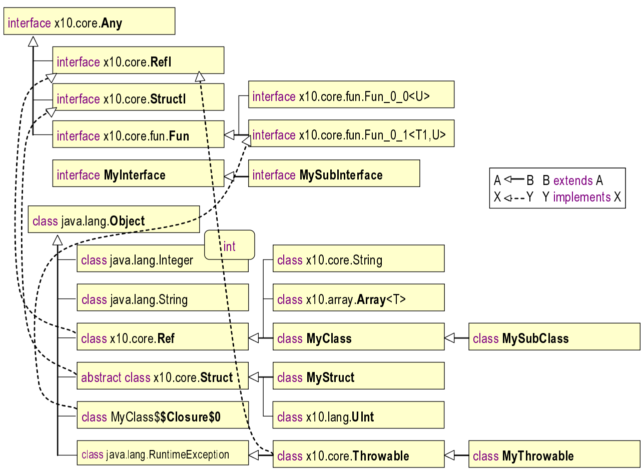


Figure 5. Type Hierarchy in Translated Java Code

2.3 Compilation and Execution

Currently, the X10 compiler is implemented as a translator into other programming languages. X10 programs can be compiled to Java [7], C++ [20], or CUDA [4] environments. The X10 environment running on Java is called *Managed X10*, while X10 in C++ is called *Native X10*. X10 code compiled to CUDA can be called from Native X10.

Figure 4 shows the flow of the compilation. The X10 compiler can roughly be divided into two parts — frontend and backends. The frontend parses X10 source code, checks types, creates AST (Abstract Syntax Trees), and performs common optimizations such as inlining. The backends generate code from the AST created by the frontend. For the Java backend, which is the main focus of this paper, it generates Java source code, which is then compiled into bytecode by a Java compiler. The generated bytecode is executed on normal JVMs with X10 runtime libraries. In Managed X10, each place corresponds to a JVM, therefore multiple JVMs are launched (typically on different nodes) to use multiple places.

Three kinds of runtime libraries are used to run X10 programs on JVMs — a common library written in X10 (XRX: X10 Runtime in X10), a backend specific library written in Java (XRJ: X10 Runtime in Java), and a common communication library written in C++ (X10RT: X10 Comm. Runtime). The X10RT [22] is used through JNI [12]. The X10 compiler itself is written in Java using the Polyglot compiler framework [14]. All of the compilers' and libraries' source code can be obtained via the X10 home page [21].

3. Representation of Types

In Managed X10, types are represented in two ways. One way is by translating X10 types to new Java classes. Another way is by mapping X10 types to the specified Java types. Figure 5 illustrates the type hierarchy in translated Java code, which corresponds to the original type hierarchy in X10 in Figure 2.

3.1 Translating X10 Types to Java Classes

Figure 6 shows how the Java backend translates X10 types to Java classes. A translated Java class has the same name as the original X10 type. Almost all of the X10 types except for some special types described in Section 3.2 can be translated with this rule.

3.1.1 Classes and Interfaces

X10 classes and interfaces are translated to Java classes and interfaces, respectively. X10 static fields are translated to Java static fields (line 39), but their initializations are performed separately, as explained in Section 7.2. Both instance fields and properties in X10 are translated to Java instance fields (lines 17–18). Properties implicitly have property methods, which are prepared as public final instance methods in Java (line 19). X10 static/instance methods are translated to Java static/instance methods (lines 20–21, 40). An X10 constructor is translated to a pair of a Java constructor and an instance field initializer (lines 27–34). This mechanism is called a two-phase constructor.

Method inlining support. The frontend implements method inlining at the AST level. Since Java backend produces Java source as a result of the compilation, the source code must be valid Java source code. For method inlining, Java access control is problematic.

For example, when inlining a method that accesses a field, the field must be accessible from the caller's body. However if the field is private and the caller belongs to a different class, the field is not accessible. Non-public classes, interfaces, methods, and fields have similar problems. Not inlining such methods is not feasible, because it would effectively prevent the inlining of all of the library methods.

To solve this problem, the Java backend translates all of the X10 classes, interfaces, static/instance fields, and static methods to Java public entities. However private instance methods cannot be translated directly to public instance methods. This is because private instance methods, which cannot be overridden, can be overridden by their subclasses if they are translated to public ones. Therefore the Java backend generates a bridge method (line 23), which is a public static method for calling the private instance method, and replaces the calls to the private instance method with calls to the bridge method in the inlined method body.

When inlining a method that calls a super class's method (line 5) into a caller whose class is different from callee's, a bridge method (line 26), which is a public instance method for calling the super class's method, is generated and the call to the super class's method is replaced with the call to the bridge method in the inlined method body.

A Java constructor creates an object and initializes its instance fields at the same time. For better performance, the instance field initialization should be inlinable, the same as instance methods. Therefore the Java backend decomposes an X10 constructor into a Java default constructor followed by an instance field initializer (lines 27–34)[1]. With this translation, the X10 constructor becomes inlinable. However, as a trade-off, its instance fields won't be final because Java does not allow initialization of final fields outside of its constructor (line 18).

Another issue in inlining is the treatment of a return value which is ignored in the caller. To avoid the generation of invalid Java code, the Java backend inserts a dummy method call to swallow the unused return value.

[1] The compilation strategy of constructor is planned to be changed in X10 2.2 to allow arbitrary statements before another constructor call.

3.1.2 Structs

X10 structs are translated to Java final classes of the same name, which implement `x10.core.StructI` interface. This is because Java does not support the struct data type of C/C++. Therefore in Managed X10, no memory will be saved even if the program is written with structs. To address this problem, we plan to translate an X10 struct into a set of variables (we call this technique struct erasure).

In Managed X10, each unsigned type (like `x10.lang.UInt`) is implemented as an X10 struct that has a field of a corresponding signed type, because Java does not support unsigned types. However, this wastes memory and lowers runtime performance. To address these problems, we plan to map unsigned types directly to corresponding signed types.

3.1.3 Functions

X10 functions are translated to Java classes that implement Java interface `x10.core.fun.{Fun,VoidFun}_0_n`, where `n` is the number of parameters and `VoidFun` does not return any value. They can be evaluated by calling their `$apply()` method with parameters.

Translating functions to static nested classes. A simple way of translating a function literal to Java is to create an instance of an anonymous subclass of `Object` that implements `{Fun,VoidFun}_0_n`, and implement the `$apply()` method in the subclass. However, since this anonymous subclass is an inner class, it captures the `this` of the outer class. If the function is passed to another place, then the `this` is unnecessarily serialized by the runtime.

To avoid this inefficiency, the Java backend translates each function literal to a static nested class that implements `{Fun,VoidFun}_0_n`. Variables captured by the function are explicitly passed to the constructor and kept as its fields. This prevents the capture and the unnecessary serialization of the `this` reference.

Translating functions to static methods. The frontend has many passes, such as method inlining and syntax-sugar handling, for transforming AST. These passes frequently generate AST nodes, each of which represents a function creation followed by an immediate invocation, like `new MyClass$$Closure$0().$apply()`.

For example, if there is a user code of the form

```
var arr:Array[Int] = ...; arr(i)+=1;
```

then the frontend transforms the syntax sugar (`+=`) to AST nodes representing the X10 code as

```
((a:Array,x:Int,v:Int)=>a(x)=a(x)+v)(arr,i,1).
```

The Java backend translates these AST nodes to a static method and its invocation, not to a function-*object* creation followed by immediate invocation. This can avoid unnecessary object creation and promote inlining by the JIT compiler.

3.2 Mapping X10 Types to the Specified Java Types

Another way to represent types in Managed X10 is by mapping X10 types to Java types. If an X10 type is declared as

```
@NativeRep("java","MyClassImpl",...)
class MyClass { ... }
```

then the X10 type `MyClass` is mapped to the specified Java class `MyClassImpl`.

This mapping mechanism is used for two reasons. One is to implement the X10 library in Java. X10 does not have a built-in mechanism to call OS functions. It is designed as an X10 standard library and its native implementation is left to the backends. In Managed X10, some X10 types that abstract OS resources (e.g. `x10.io.File.NativeFile`) are mapped to corresponding Java native implementations.

The other reason is to improve performance by using Java primitives and well-known types. A JVM can handle primitive types (such as `int`) more efficiently than objects. A JVM also implements special optimizations for well-known types (such as `String`) based on built-in knowledge. Performance is improved when Managed X10 maps X10 types to Java primitives or well-known types. Also by mapping X10 types to Java types that are used as parameters or return values by Java APIs, we can call them without translating the parameters and return values. Table 1 summarizes the X10 types that are mapped to primitive or well-known Java types.

X10 Code

```
class C(p:Int) {
  val q:Int; var r:Int;
  def u() { }
  private def v() { }
  def w() { return super.hashCode(); }
  def this(p:Int) {
    property(p);
    q = 1;
  }
  static val s = Place.MAX_PLACES;
  static def t() { }
}
```

Translated Java Code

```
import x10.lang.Place;
import x10.lang.Runtime;
import x10.runtime.impl.java.InitDispatcher;
public class C extends x10.core.Ref {
  public int p;                  // property
  public int q, r;               // instance fields
  final public int p() { return this.p; } // property method
  public void u() { }            // instance methods
  private void v() { }
  // bridge method for private instance method
  public static void v$P(final C C) { C.v(); }
  public int w() { return super.hashCode(); }
  // bridge method for super.hashCode()
  final public int C$hashCode$S() { return super.hashCode(); }
  public C(final int p) {        // constructor
    super();
    this.p = p;
    this.__fieldInitializers173();
    this.q = 1;
  }
  // instance field initializer
  final private void __fieldInitializers173() { this.r = 0; }
  // bridge method for field initializer
  final public static void __fieldInitializers173$P(final C C) {
    C.__fieldInitializers173();
  }
  public static int s = 0;       // static field
  public static void t() { }     // static method
  // static initialization
  public static int fieldId$s; // static field id
  // static field initializer
  public static int getInitialized$s() {
    if (Runtime.hereInt() == 0) {
      C.s = Place.getInitialized$MAX_PLACES();
      InitDispatcher.broadcastStaticField(C.s, C.fieldId$s);
    }
    return C.s;
  }
  // static field deserializer
  public static void getDeserialized$s(byte[] buf) {
    C.s = (Integer) InitDispatcher.deserializeField(buf);
  }
  static {                       // Java static initializer
    C.fieldId$s = InitDispatcher.addInitializer("C", "s");
  }
}
```

Figure 6. X10 Class and Translated Java Class

Numbers, Character and Boolean. In X10, Numbers (`x10.lang.Int` etc.), Character (`x10.lang.Char`), and Boolean (`x10.lang.Boolean`) are defined as structs. For better performance, these types are mapped to Java primitives (`int` etc., `char`, and `boolean`). When casting these types to `Any` or a parameter type `T`, the Java compiler generates boxing code for their wrapper classes (`java.lang.Integer` etc., `java.lang.Character`, and `java.lang.Boolean`).

String. The string class in X10 is `x10.lang.String`. For performance reasons, this is mapped to `java.lang.String`. When casting `x10.lang.String` to `Any`, `Object`, or a parameter type `T`, the X10 compiler generates boxing code for its native implementation `x10.core.String` that extends `x10.core.Ref` and implements `Fun_0_1<Integer,Character>`. This boxing is also performed when a string is cast to the function `(Int)=>Char`, which is implemented by `x10.lang.String`, to get a character at the specified position.

X10	Java
x10.lang.Boolean	boolean (java.lang.Boolean when boxed)
x10.lang.Char	char (java.lang.Character when boxed)
x10.lang.Byte	byte (java.lang.Byte when boxed)
x10.lang.Short	short (java.lang.Short when boxed)
x10.lang.Int	int (java.lang.Integer when boxed)
x10.lang.Long	long (java.lang.Long when boxed)
x10.lang.Float	float (java.lang.Float when boxed)
x10.lang.Double	double (java.lang.Double when boxed)
x10.lang.String	java.lang.String (x10.core.String when boxed)
x10.lang.Comparable[T]	java.lang.Comparable<T>
x10.lang.Throwable	x10.core.Throwable
x10.lang.Object	x10.core.RefI
(any structs)	x10.core.StructI
x10.lang.Any	java.lang.Object

Table 1. X10 Types Mapped to Java Primitives or Well-known Java Types

Comparable. X10 interface `x10.lang.Comparable[T]` is mapped to Java interface `java.lang.Comparable<T>`. With this mapping, X10 types that implement `x10.lang.Comparable[T]` can be mapped to Java classes that implement `java.lang.Comparable<T>`. This technique is used for string, character, boolean, and all of the numeric types except for unsigned types.

Throwable. Unlike Java, X10 does not have checked exceptions nor any `throws` clause. To implement the X10's exception model efficiently, we decided to make every X10 exceptions Java runtime exception. Managed X10 maps the top-level exception type, `x10.lang.Throwable`, to `x10.core.Throwable` which extends `java.lang.RuntimeException`. To handle Java checked exceptions thrown by Java native libraries, the Java backend generates an enclosing try-catch block for each call that may cause Java checked exceptions, and wraps each caught exception with `x10.runtime.impl.java.WrappedThrowable` that extends `x10.core.Throwable`.

Any and Object. `x10.lang.Object` is mapped to `x10.core.RefI` that is a marker interface must be implemented by all of the X10 classes in translated Java code. This means we need a wrapper class (e.g. `x10.core.String`), which implements `RefI`, for each raw Java class (`java.lang.String`) when it is mapped to X10 class (`x10.lang.String`).

`x10.lang.Any` is mapped to `java.lang.Object`. This is because X10 variables of type `Any` must be able to hold the value of arbitrary X10 types, and some of the X10 types are mapped to existing Java types. Interface methods of `Any` (e.g. `typeName()`) are specially handled by Java backend and translated to runtime calls.

4. Implementation of Generics

For each X10 class with generics, Managed X10 generates a Java class with Java's generics (Figure 7). However, there is a semantic gap between X10's generics and Java's generics. In this section, we describe how Managed X10 bridges the gap without severe performance degradation.

4.1 Implementing X10 Generics on Java

X10 generics are implemented with type reification that keeps the type parameters, unlike Java generics, which are implemented with type erasure that removes the type parameters after the type check. Because type parameters are required to process `as`, `instanceof`, `<:`, `:>`, and `typeName()`, Java objects generated from X10's parameterized types have their own type parameters.

In addition, X10 allows a type to implement multiple interfaces based on the same base type. However, Java doesn't allow these types because Java can't distinguish the methods in two interfaces based on the same base type after its type erasure. To implement these types in Java, Managed X10 generates dispatch methods to distinguish between these methods.

X10 Code

```
1  interface I[T] {
2    def m(T):T;
3  }
4  class B[T] {
5    def m(a:T):T {return a;}
6  }
7  class C[T1,T2] extends B[Int] implements I[String], I[Int] {
8    def this(T1){}
9    def this(T2){}
10   def m(a:T1){return a;}
11   def m(a:T2){return a;}
12   public def m(a:String){return a;}
13   public def m(a:Int){return a;}
14 }
```

Translated Java Code

```
15 import x10.rtt.RuntimeType;
16 import x10.rtt.Type;
17 import x10.rtt.Types;
18 interface I<T> {
19   public static final RuntimeType<I> $RTT = ...
20   Object m(final T id$0, Type t1);
21 }
22 public class B<T> extends x10.core.Ref {
23   public static final RuntimeType<B> $RTT = ...
24   private Type T;
25   public T m_0_$$B_T$G(final T a) {return a;}
26 }
27 public class C<T1, T2> extends B<Integer> implements I {
28   public static final RuntimeType<C> $RTT = ...
29   private Type T1, T2;
30   // dispatcher for abstract public I.m(id$0:T):T
31   public Object m(final Object a1, final Type t1) {
32     if (t1.equals(Types.STRING)) {
33       return m((String) a1);
34     } else if (t1.equals(Types.INT)) {
35       return m((int)(Integer) a1);
36     }
37     return null;
38   }
39   // bridge for B.m(a:T):T
40   public Integer m_0_$$B_T$G(Integer a1) {return m((int) a1);}
41   // constructors need signature mangling
42   public C(final Type T1, final Type T2,
43                final T1 id$1, Class $dummy0) {...}
44   public C(final Type T1, final Type T2,
45                final T2 id$2, Class[] $dummy0) {...}
46   // generic methods need signature mangling
47   public T1 m_0_$$C_T1$G(final T1 a) {return a;}
48   public T2 m_0_$$C_T2$G(final T2 a) {return a;}
49   // instantiated generic methods
50   public String m(final String a) {return a;}
51   public int m(final int a) {return a;}
52 }
```

Figure 7. X10 Generics

4.1.1 Representation of Runtime Type Information

For each Java class created from an X10 class, there is a static field named `$RTT`, whose type is `x10.rtt.RuntimeType` (lines 19, 23, and 28). This field holds all of the static information for the corresponding X10 class. For example, the information that class `C` implements both `I[String]` and `I[Int]` is held in its `$RTT` (line 28).

Since X10 generics are implemented with type reification, each Java object created from a generic X10 class has additional fields whose type is `x10.rtt.Type`, which is used to hold actual values of the type parameters (`T1` and `T2` in line 29).

When an X10 generic class is instantiated, objects of `x10.rtt.ParameterizedType` or `RuntimeType` are stored into the instance fields. If a `ParameterizedType` object is stored, the corresponding parameter type is a parameterized type. Otherwise, the parameter type is a type that doesn't have any parameter type.

For example, for `C[Point,Array[Point]]`, `Point.$RTT`, whose type is `RuntimeType`, is stored into the first parameter type (`T1`). Also, a new `ParameterizedType` object is stored into the second (`T2`). This `ParameterizedType` object has `Array.$RTT` object as its base type and `Point.$RTT` as its type parameter.

4.1.2 Signature Mangling

When a class has a method (or a constructor) using type parameters for its arguments (e.g. `m(T1)` at line 10 or `this(T1)` at line 8 in Figure 7), X10 allows another method (or constructor) of the same signature with different type parameters for the arguments (such as m(T2) at line 11 or this(T2) at line 9), but Java doesn't. In Managed X10, these methods and constructors are mangled and the corresponding call sites are also changed to call the mangled methods and constructors.

For methods using type parameters for their arguments, the names of the parameter types are added to the method names. Also, the call sites are changed to call the changed methods. In Figure 7, Managed X10 generates m_0_$$C_T1$G(T1) (line 47) and m_0_$$C_T2$G(T2) (line 48) for m(T1) (line 10) and m(T2) (line 11).

For constructors using type parameters for their arguments, their signatures are changed by adding dummy arguments like java.lang.Class and java.lang.Class[]. Also, the call sites are changed to call the changed constructors. In Figure 7, C(Type, Type,T1,Class) (lines 42–43) and C(Type,Type,T2,Class[]) (lines 44–45) are generated for this(T1) (line 8) and this(T2) (line 9) [2].

4.1.3 Self Dispatch

X10 allows a type to implement multiple interfaces based on the same base type with different type parameters, but Java doesn't. The technique of method mangling doesn't bridge this semantic gap because the mangled signatures can't be determined at compilation time for the interface of this base type. Therefore, in Managed X10, if an interface has a method using type parameters for its arguments, this method is translated to a dispatch method and generates its implementations in the classes implementing the interface. This dispatch method is generated by adding Type arguments to the original. Each Type argument is mapped to an argument of a parameter type in the original method and each call site for the method specifies the Type objects corresponding to its arguments. For example, I[T].m(T) (line 2) in X10 is translated to I<T>.m(T, Type) (line 20) in Java. This method is implemented in C<T1,T2>, which implements I for I[String] and I[Int], to dispatch m(String) and m(int) (lines 31–38). At the call site of the dispatch method, x10.rtt.Types.STRING is specified for the I[String].m(String) call and x10.rtt.Types.INT is specified for the I[Int].m(Int) call as the second argument. The Types.STRING and Types.INT are the runtime types for x10.lang.String and x10.lang.Int, respectively.

4.2 Performance Optimization by Generating Bridge Methods

X10 allows specifying primitive types as the parameter types, but Java doesn't. Therefore, primitive types for the parameter types in X10 are translated to the corresponding wrapper classes in Java such as B<Integer> (line 27) for B[Int] (line 7). However, boxing for parameter types causes override misses. For example, B[T].m(T) (line 5) is overridden by C[T1,T2].m(Int) (line 13) in X10 because C[T1,T2] extends B[Int]. In contrast, B<T>.m(T) is never overridden by C<T1,T2>.m(int) in Java, because B<T>.m(T) becomes B.m(Object) through type erasure.

In Managed X10, bridge methods are generated to override methods using type parameters for their arguments. Initially, these methods are mangled by changing the method names. Next, bridge methods are generated in the classes overriding the mangled methods. For B[T], B[T].m(T) is mangled to B<T>.m_0_$$B_T$G(T) (line 25). This B<T>.m_0_$$B_T$G(T) is overridden by C<T1,T2>. m_0_$$B_T$G(T) (line 40) that dispatches to C<T1,T2>.m(int) (line 51).

A method using a type parameter for its return value is always mangled to allow subclasses to override the method and implement dispatch. For example, get():T in X10 is mangled to T get$G() in Java. If a method get():T of a class is overridden by get():Int of its subclass in X10, a dispatch method Object get$G() is generated in the subclass in Java.

[2] Class can be replaced with any Java class that does not conflict with all of the valid X10 types in the translated Java code.

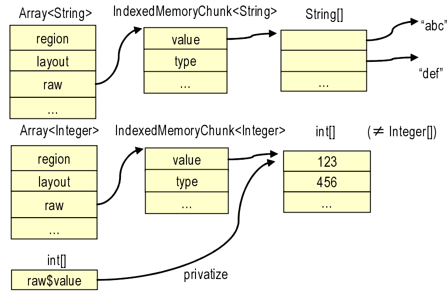


Figure 8. Object Model of Array

To reduce the overheads of dispatches, theses mangled methods are called only when the types of their arguments aren't determined at compilation time. Otherwise, the target methods for the dispatches are called directly by the caller. Also, if the argument types of these methods are primitive types, then their caller calls them without boxing.

5. Array Optimizations

In X10, arrays are represented with a generic class x10.array. Array[T] [3]. Since Array[T] is designed as a general array type that can represent multi-dimensional or sparse arrays, its implementation is decomposed into two parts, a one-dimensional contiguous backing array which is indexed with a single integer offset, and the offset calculation mechanism. The backing array is represented with x10.util.IndexedMemoryChunk[T], which is mapped to Java generic class x10.core.IndexedMemoryChunk<T>. IndexedMemoryChunk<T> holds the array data as a Java array (Figure 8). Accessing the contents in the Java array is done by calling the $set() and $apply() methods of IndexedMemoryChunk<T>.

5.1 Footprint Reduction

Java does not allow instantiating a parameter type T with primitive types such as int, and therefore if we use T[] for the type of the Java array in IndexedMemoryChunk<T>, it is instantiated as Integer[]. Since this means X10 arrays use more memory than Java arrays for primitive types, we don't use this approach but map them to Java primitive arrays. To make this possible, we use Object for the type of the field that holds the Java array in IndexedMemoryChunk<T>. Creations of and accesses to the Java arrays are implemented with utility functions defined in x10.rtt.Type interface. Type.makeArray() creates a Java array that corresponds to its base type, and setArray() and getArray() access the array elements by casting the field of Object type to the appropriate Java array type.

5.2 Access Inlining

Unfortunately, a naive implementation of this mechanism performs poorly. This is because a method invocation, a field access, and a dynamic cast are required for each array access. To make matters worse, boxing and unboxing occurs for the elements of the primitive arrays.

In order to prevent these boxings and unboxings, when the actual type of the array is known, the Java backend inlines $set() and $apply() of IndexedMemoryChunk<T>, and generates code that accesses that Java array directly (line 23 of Figure 9).

[3] X10 also has x10.lang.Rail[T] that represents a zero-origin contiguous one-dimensional array. Because this can easily be compiled to a Java array, it is heavily used with performance critical applications in early implementations of X10. Now the X10 compiler translates Array[T]{rail} to equivalent code as Rail[T], so that it is no longer needed.

```
X10 Code
val a:Array[Int] = ...;
val p:Point(a.rank) = ...; // p has the same rank as a
... = a(p);
Translated Java Code
final x10.array.Array<Integer> a = ...;
final x10.array.Point p = ...;
final x10.array.Array<Integer> this443 = a;
final x10.array.Point pt444 = p;
int ret438445 = 0;
__ret439446:
do {
  // array bound check
  final x10.array.Region t493 = this443.region;
  final boolean t494 = t493.contains(pt444);
  if (!t494) {
    x10.array.Array.raiseBoundsError$P(pt444);
  }
  // offset calculation
  final x10.array.RectLayout t497 = this443.layout;
  final int t499 = t497.offset(pt444);
  // array data access
  final x10.core.IndexedMemoryChunk<Integer> t498
    = this443.raw;
  final int t500 = ((int[]) t498.value)[t499];
  ret438445 = t500;
  break __ret439446;
} while (false);
... = ret438445;
```

Figure 9. Array Access Inlining

```
X10 Code
1 val arr:Array[Int]{rail} = ...;
2 val raw = arr.raw();
3 var sum:Int = 0;
4 for (var i:Int = 0; i < arr.size; i++) {
5   sum += raw(i);
6 }
Translated Java Code
7 final x10.array.Array<Integer> a = ...;
8 final x10.array.Array<Integer> this601 = a;
9 final x10.core.IndexedMemoryChunk<Integer> raw = this601.raw;
10 int sum = 0;
11 // privatize Java array
12 final int[] raw$value620 = (int[]) raw.value;
13 for (int i = 0; i < arr.size; i = i + 1) {
14   sum = sum + raw$value620[i];
15 }
```

Figure 10. Privatization of a Java Array

5.3 Privatization of Java Array

When an array is accessed in a loop a field access and a dynamic cast are executed at each loop iteration, which are redundant since they produce the same results. To eliminate this redundancy, the Java backend privatizes the field of the `Object` type that holds the Java array as the actual Java array type, and hoists it out of the loop (line 12 of Figure 10).

We have already implemented this optimization for `Rail` and `IndexedMemoryChunk`, and are currently working on it for `Array`.

6. Parallel/Distributed Execution

Language-level support of parallel and distributed execution is one of the features that differentiate X10 from other languages. In this section, we describe how Managed X10 implements these important features.

6.1 Asynchronous Execution

In X10, an asynchronous activity is created by a statement `async S`, where `S` is the statement to be spawned. For each `async` statement, a function of type `()=>void` that executes `S` is created and specified as the argument for the runtime helper method `x10.lang.Runtime.runAsync()`. The `runAsync()` method creates a `x10.lang.Activity` object holding the specified function as its field. The created `Activity` is specified for the X10's work-stealing mechanism that is similar to Java Fork/Join [11].

In Managed X10, each generated function is declared as the static nested class in the spawning class. This class implements the `VoidFun_0_0` interface and its `$apply()` method contains the statements translated from `S`. For the example of the X10 program (lines 1–3 of Figure 11), the static nested class `$Closure$0` is declared (lines 10–17) corresponding to the `async` statement (line 3).

```
X10 Code
1 var m = 0;
2 val i = 1;
3 async { m = i; }
Translated Java Code
4 int m = 0;
5 final int i = 1;
6 final int[] $m173 = new int[1]; $m173[0] = m;
7 x10.lang.Runtime.runAsync(new $Closure$0($m173, i));
8 m = $m173[0];
9 ...
10 public static class $Closure$0 extends x10.core.Ref
11  implements x10.core.fun.VoidFun_0_0 {
12   public int[] $m173; public int i;
13   ...
14   public void $apply() { $m173[0] = i; }
15   public $Closure$0(final int[] $m173, final int i) {
16     this.$m173 = $m173; this.i = i;
17 }}
```

Figure 11. Asynchronous Execution

```
X10 Code
1 var m = 0;
2 val i = 1;
3 val h = here;
4 at (here.next()) {
5   at (h) m = 2;
6 }
Translated Java Code
7 import x10.core.LocalVar;
8 import x10.lang.Place;
9 import x10.lang.Runtime;
10 int m = 0;
11 final int i = 1;
12 final Place h = Runtime.home();
13 final LocalVar<Integer> m$b
14  = new LocalVar<Integer>(Types.INT, m, null);
15 Runtime.runAt(Runtime.home().next(),
16  new $Closure$2(i, h, m$b, null));
17 m = m$b.$apply$G();
18 ...
19 public static class $Closure$1 extends x10.core.Ref
20  implements x10.core.fun.VoidFun_0_0 {
21   public LocalVar<Integer> m$b;
22   ...
23   public $Closure$1(final LocalVar<Integer> m$b, Class $d) {
24     this.m$b = m$b;
25   }
26   public void $apply() {
27     int m = m$b.get$G(); m$b.set$G(m = 2);
28 }}
29 public static class $Closure$2 extends x10.core.Ref
30  implements x10.core.fun.VoidFun_0_0 {
31   public int i; public Place h; public LocalVar<Integer> m$b;
32   ...
33   public $Closure$2(final int i, final Place h,
34    final LocalVar<Integer> m$b, Class $dummy0) {
35     this.i = i; this.h = h; this.m$b = m$b;
36   }
37   public void $apply() {
38     x10.io.Console.getInitialized$OUT().println(i);
39     Runtime.runAt(h, new $Closure$1(m$b, null));
40 }}
```

Figure 12. Place Change

This `$Closure$0` is instantiated whenever the activity is spawned by calling the `runAsync()` method (line 7). In the `runAsync()` method, an `Activity` object for the `$Closure$0` object is generated and enqueued in the queue of the X10's workstealing mechanism.

6.2 Place Change

X10 supports the execution of a statement in a different place with the `at` statement, such as `at (p) S`, where `S` is the statement to be executed and `p` is the place where the statement `S` will be executed. The caller of an `at` statement will be blocked until the `at` statement returns. For each `at` statement, a function of type `()=>void` that executes statement `S` is created and specified as the argument for the runtime helper method `x10.lang.Runtime.runAt()`. The `runAt()` method serializes the specified function, sends the serialized data to place `p`, and calls the `$apply()` method for the function in the place `p`. The `p` may be a local or remote place, and even in for a local place (i.e. `p == here`), the specified function will be serialized. For the example of the X10 program (lines 1–6 of Figure 12), the static nested class `$Closure$2` is declared (lines 29–40) corresponding to the `at` statement (line 4). The `$Closure$2` is instantiated whenever the at statement (line 4) is executed by calling runAt() method (lines 15–16).

Managed X10 serializes functions by using the Java default serializer [9]. For better performance and better interoperability with other backends, we plan to generate a custom serializer for each X10 type.

6.3 Local Variable Access in async/at Body

An async or at statement is allowed to reference val and var local variables declared in the lexically enclosing blocks. Also, the statement is allowed to update var local variables declared in the lexically enclosing blocks and the updated value becomes available when the program reaches the end of the finish block that encloses the async statement or the end of the at statement. In the generated Java code, however, these variables cannot be referenced because this statement is translated to the $apply() method of a function of type ()=>void.

In Managed X10, to reference these local variables in a function for an async or at statement, they are specified as the function's constructor arguments. Also, to update these local variables in an async or at statement, they are copied to the heap by boxing and specified as the function's constructor arguments. These specified values are referenced and updated in the $apply() of the function. In addition, the boxed values are copied back to the local variables at the end of the finish block for the async statement or at the end of the at statement.

The boxing of the local variables for async and at are different. For async, each local variable is boxed as a Java array that has only one element (line 6 of Figure 11). When instantiating the function of an async (line 7), the Java array is created and the local variable is set to the 0-th element of the array (line 6). Also, when updating the variable to a new value in the async statement, the new value is set to the 0-th element of the array (line 14). Finally the boxed value is copied back to the local variable (line 8).

For the at statement, each local variable is boxed to an x10.core.LocalVar<T> object (lines 13–14 of Figure 12). This LocalVar object has an ID. Each place has mappings from each ID to the object that is referenced by the LocalVar object of the ID. At instantiating the function of an at (lines 15–16), an ID is assigned to the local variable and the referenced object is mapped to the ID. Also, at updating the variable to a new value in the at statement, the new value is mapped to the ID by calling the LocalVar.set$G() method (line 27). Finally the boxed value is copied back to the local variable (line 17).

6.4 Global Reference

An x10.lang.GlobalRef[T] struct is a reference to an object in an arbitrary place. The equality between two GlobalRef structs is guaranteed in any places when they reference the same object.

In Managed X10, a GlobalRef[T] struct is mapped to an x10.core.GlobalRef<T> object that has the place field of the x10.lang. Place type and the id field of the long type. place represents the place where the object that is referenced by the GlobalRef struct is created and id is the ID for the object. When two place and two id of two GlobalRef are same, they reference the same object in the same place.

The ID for an object is assigned when the object is referenceable by GlobalRef in some other place. Actually, the ID is assigned in the serializer of GlobalRef only when no other ID is assigned to the referenced object.

Each place manages mappings from an object to its ID (id2Object) and from an ID to the object (object2Id) for all of the objects generated in that place. When a program tries to get the object that is referenced by a GlobalRef struct, the program calls the GlobalRef.$apply() method that returns the object from the id2Object mapping with the id of the GlobalRef. Also, when a program creates a GlobalRef struct that references an object and serializes the GlobalRef, the program assigns a new ID to the referenced object only when object2Id doesn't

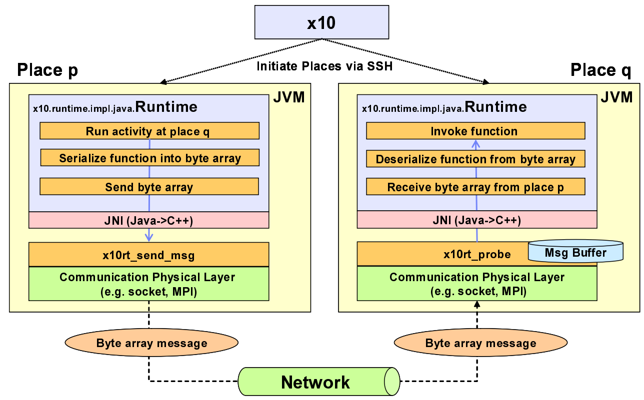


Figure 13. Distributed Execution Framework

contain the ID of the object. These mappings are contained in two static java.util.concurrent.ConcurrentHashMap objects and the operations for these mappings never block any others (non-blocking registration).

If an object in a place is referenced by GlobalRef structs in other places, the object cannot be collected by the GC because the static ConcurrentHashMap objects contain references to the object. However, there are some data types that are guaranteed not to be referenced from other places at some locations in a problem. For example, an x10.lang.FinishState object, which is created for each finish in XRX, is referenced by some GlobalRef structs in other places to notify the end of the activities in the finish. Because it is not necessary to notify anything after the finish, the FinishState object will never be referenced by any GlobalRef structs after the end of the corresponding finish. This type of object can be collected by the GC locally in each place if the program specifies the location where the object can be collected.

X10 provides the x10.lang.Runtime.Mortal interface to specify when an implementing object can be collected. If an object implements this interface, the GC is allowed to collect the object when it isn't referenced by any objects in the place. In Managed X10, if an object implements Mortal, the object is registered in id2Object and object2Id through java.lang.ref.WeakReference to implement these semantics.

7. Multi-JVM Support

This section describes the distributed execution feature that was introduced in Managed X10 2.1.2.

7.1 Distributed Execution Framework

Managed X10 is also designed as a scalable platform on a cluster of nodes. The design philosophy is that one place corresponds to one JVM process.

Figure 13 shows the distributed execution framework of Managed X10. The X10 runtime has a language-independent communication layer that can be used in both the C++ and Java backends. The layer provides a unified API for communication between multiple places by abstracting out a wide variety of concrete communication libraries such as MPI [13], PGAS [15], raw sockets, and others. By providing such a high-level communication layer, it can easily be replaced with an appropriate communication library optimized for the underlying execution platform. If a scientific application needs a large number of message exchanges between places on a large cluster of nodes, an optimized MPI library can be used. Then all of the high-level communication layer is mapped to the appropriate functions provided by MPI. In contrast, the raw socket library has been developed from scratch and is used by default unless users explicitly specify a communication library in the X10 compiler options. This is more convenient for users since they need not install or configure any communication libraries if their application does not require an optimized communication layer. The current version of the Java backend runtime has been developed to employ a socket-based library, but it could be extended to other communication libraries, which has already been done for C++ backend runtime.

The invocation mechanism for spawning off an X10 runtime at a remote place depends on which library is used. For MPI, the command tool `mpirun` handles it via SSH. The socket-based library uses the `x10` command that also uses SSH for remote invocation. Invoking a function at a remote place is executed in 4 steps: (1) a function object is serialized into a byte sequence, (2) it is transferred to the remote place with the specified communication layer, (3) the byte sequence is deserialized into the function object, and (4) then the `$apply()` method of the function object is invoked.

7.2 Static Initialization

X10 guarantees each static field has the same value in all of the places in a program run.

Java initializes static fields with a static initializer that is executed at class loading time. A JVM usually loads classes just before the program uses them. X10 programs usually run asynchronously in multiple places, and therefore if we use the Java static initializer to initialize the X10 static fields, they may have different values at different places. It is not realistic to load classes in the same order or to control the timing of class loading in different places to avoid this kind of problem. Therefore Managed X10 implements the initialization of static fields by initializing the static fields in `Place.FIRST_PLACE` (i.e. the place where the program started running), and broadcasts them to all of the other places using these steps:

1. Load all of the Java classes that were translated from the X10 types and that are reachable from the application's main class by using Java reflection.
   The Java static initializer (lines 55–57 of Figure 6) of the classes registers a static field initializer (lines 44–50) and a deserializer (lines 52–54) for each static field (line 39).
2. Translate all of the registered static field initializers to X10 functions and execute each of them asynchronously.
   If the current place is `FIRST_PLACE`, then the static field initializer
   (a) initializes the static field
   (b) serializes the field, creating a function that will execute the static field deserializer with the serialized data, and executes it in all of the places except for `FIRST_PLACE`.
   The static field deserializer deserializes the data and stores the value in the static field.

This implementation has some problems related to class preloading. For example, the memory usage may be larger than needed because of unnecessary class loading. Also, in some JVM implementations, loading many classes in a short period of time may cause the JIT compiler to use a lower optimization level for quicker application initialization, and thus result in lower performance of the generated code or a longer time to achieve stable performance. To solve these problems, we are investigating the possibility of lazy class loading in X10.

8. Performance Evaluation

Table 2 is a summary of the major performance features which implemented in Managed X10 since 2010. In this section, we present our performance improvements with sequential and parallel (`async`) benchmarks. We also present initial scalability result with a distributed benchmark. All measurements were done on a multicore node that has two sockets for 2.93-GHz Intel Xeon X5670 chips with a total of 12 physical cores with SMT turned off and 16 GB of RAM running 64-bit Red Hat Enterprise Linux Server release 5.5 (kernel 2.6.18-194.el5) and IBM J9 VM (build 2.4, JRE 1.6.0 IBM J9 2.4 Linux amd64-64 jvmxa6460sr9-20110203_74623 (JIT enabled, AOT enabled)).

Release	Date	Performance Features
2.0.3	April 17, 2010	(base release)
2.0.4	June 14, 2010	Array(IndexedMemoryChunk) access inlining Bridge method for Java primitive types
2.0.5	July 23, 2010	Privatize backing Java array in loops
2.0.6	September 3, 2010	Function to static method
2.1.0	October 19, 2010	(New object model)
2.1.1	January 10, 2011	(Static initialization)
2.1.2	February 25, 2011	Multi-JVM Method inlining GlobalRef (non-blocking and lazy registration) Function to static nested class Use default optlevel while preloading

Table 2. Major Performance Features in Managed X10

8.1 Sequential Performance

We have been improving sequential execution in X10. We first looked at the performance of `Rail`, because it was used to represent array data before X10 release 2.1.0. `Rail` is now deprecated in favor of `Array`, thus we are focusing on the performance of `Array`.

Figure 14(a) shows the performance of `KMeansSequential`, which is a benchmark for calculating K-means clustering that divides 1,000,000 four-dimensional points into 4 clusters. We compared the shortest elapsed times of 10 executions with Managed X10 and Java. All evaluations were done at a single place.

The performance of the `Rail` version of the benchmark was comparable to Java. The performance of the `Array` version was improved in X10 release 2.1.2, but there is still a gap between Managed X10 and Java. We are studying the overhead in our generated Java code.

8.2 Parallel Performance

We have also been improving the asynchronous execution in X10, as well as the sequential execution. For each `finish`, a `FinishState` object and a `GlobalRef` struct for the `FinishState` are generated. To optimize asynchronous execution in X10, the runtime should reduce the overhead for the `GlobalRef` mechanism that registers referenced objects by assigning an ID to each object. We introduced two optimizations to reduce overhead: eliminating synchronized blocks for the registration (non-block registration) and avoiding registration for non-escaping objects (lazy registration).

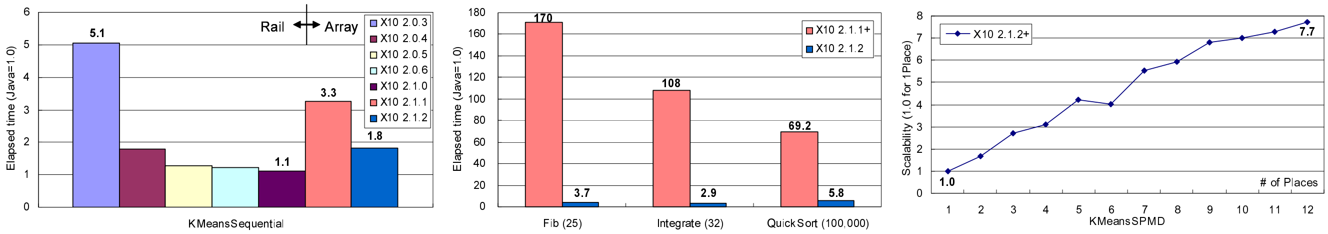
Figure 14(b) shows the performance results with these optimizations. We used three benchmarks, `Fib`, `Integrate` and `QuickSort` in the samples/work-stealing directory, each of which spawns a large number of statements with `async`. In each run of these benchmarks, the elapsed time for the run with the specified parameter is measured 10 times. We compared the average of last 5 scores with the results of the equivalent benchmarks written with Java Fork/Join. All evaluations were done at a single place.

As shown in Figure 14(b), the asynchronous performance has been improved with our optimizations. However, there is still a gap between Java Fork/Join and X10. We are adding more optimizations to Managed X10 for asynchronous execution.

8.3 Scalability with Multi-JVM

We evaluated the scalability of distributed execution in Multi-JVM-enabled X10. For the target application, we used `KMeansSPMD` in the samples directory, which calculates distributed K-means clustering that divides 20,000,000 two-dimensional points into 500 clusters. We measured the median elapsed time of 10 executions for each number of places. All evaluations were done at multiple places on a single node.

Figure 14(c) shows the performance with varying numbers of places, where the vertical axe of the graph indicates the speed-up from 1 place. The results shown in the graph demonstrate good scalability, since the speed-up ratio against 1 place is increasing



(a) Sequential benchmark results (b) Parallel benchmark results (c) Distributed benchmark results

Figure 14. Benchmark Results

with more places up to 12, and reaches a maximum of 7.7 with 12 places. For more better scalability, we are investigating the overhead in our Multi-JVM implementation.

9. Related Work

As we are entering more deeply into the multicore era, there has been increasing demand for new models and languages that simplify application development for clusters of multicore machines. While some proposals address specific domains, such as MapReduce [5] for data processing and SPL [8] for stream computing, general-purpose languages have also been proposed, including X10 [21], Chapel [2], Fortress [16], and Go [6].

Implementing programming languages by using a Java virtual machine (JVM) is a common approach. These languages are called JVM languages, including, existing languages, such as JRuby [10] and the IBM implementation of PHP [17], and new languages, such as X10 [21] and Scala [19]. In almost every attempt to create a JVM language, although some constructs are mapped to the JVM in a straightforward manner, others need to deal with subtle differences (such as generics in X10) and complete omissions (such as closures in X10).

The reference implementation of X10 provides a Java backend, described in this paper, and a C++ backend, intended to appeal to both Java and C++ programmers. There are fewer language systems that attract both of the Java and C++ camps.

10. Conclusion

In this paper, we discussed various compilation techniques for implementing advanced X10 features that cannot be mapped directly to Java without severe performance loss. In X10 release 2.1.2, by using appropriate implementations, sequential performance has been improved by about 5 times and is now comparable to Java. We are now focusing on improving the performance of the general Array class. Parallel performance has also been improved with an optimized activity creation mechanism and the gap to Java Fork/Join performance is about 3 times when run at a single place. Initial evaluation of distributed execution shows good scalability.

Some compilation techniques, such as bridge methods for calling private instance method or super class's method, are specific to Java backend that generates Java source code. They are not needed for the backend that generates Java bytecode directly, however it is difficult to develop and maintain such backend for an evolving language, such as X10, since it takes longer time to update the bytecode backend to catch up with the rapidly changing specification.

We believe that the compilation techniques described in this paper can also be used for implementing other programming languages targeted for Java or other managed environments.

Acknowledgments

We would like to thank the core members of the X10 compiler team, Vijay Saraswat, David Grove, Igor Peshansky, and Olivier Tardieu, for their initial implementation and regular discussions on the design of Managed X10.

References

[1] G. Bracha. Generics in the Java Programming Language. `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

[2] The Chapel Parallel Programming Language. `http://chapel.cray.com/`

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pp. 519–538, 2005.

[4] CUDA C Programming Guide, Version 3.2. `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, 2004.

[6] The Go Programming Language. `http://golang.org/`

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, Third Edition. Addison-Wesley Publishing Company. 2005.

[8] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu. SPL Stream Processing Language Specification. IBM Research Report, RC24897, 2009.

[9] Java Object Serialization Specification. `http://download.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html`

[10] JRuby. `http://www.jruby.org/`

[11] D. Lea. A Java Fork/Join Framework, In *ACM Java Grande 2000 Conference*, pp. 36–43, 2000.

[12] S. Liang. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley Publishing Company. 1999.

[13] MPI: A Message-Passing Interface Standard, Version 2.2. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`

[14] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC '03)*, LNCS 2622, pp. 138–152, 2003.

[15] PGAS – Partitioned Global Address Space Languages. `http://www.pgas.org/`

[16] Project Fortress Community. `http://projectfortress.sun.com/`

[17] Project Zero: PHP on Java. `http://www.projectzero.org/php/`

[18] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf`

[19] The Scala Programming Language. `http://www.scala-lang.org/`

[20] B. Stroustrup. The C++ Programming Language, Third Edition. Addison-Wesley Publishing Company. 1997.

[21] X10 Home. `http://x10-lang.org/`

[22] X10RT API Specification. `http://dist.codehaus.org/x10/x10rt/`