International Journal of Parallel Programming (2021) 49:911–933
https://doi.org/10.1007/s10766-021-00719-w

Fortress Abstractions in X10 Framework

Anshu S. Anand[1] · Karthik Sayani[2] · R. K. Shyamasundar[3]

Received: 14 November 2020 / Accepted: 8 June 2021 / Published online: 15 July 2021

Abstract

Fortress provides a nice set of abstractions used widely in scientific computing. The use of such abstractions enhances the productivity of programmers/users. Also, in scientific computations, boilerplate code has extensive usage. Keeping this in view, we embed Fortress abstractions in an X10 environment so that we can get better productivity without losing performance. In this paper, we transform Fortress into X10 through a transcompilation system. We describe compilation strategies for a few important constructs and discuss the performance of the generated X10 code with respect to the original Fortress code. The translated X10 code outperforms the original Fortress code with a maximum of 206x speedup achieved in the best case. The system also supports the multiresolution language approach that simplifies parallel programming by allowing domain scientists to write programs in the Fortress syntax that is closer to the mathematical notation. The translated X10 code, which can further be compiled to either C++ or Java, implicitly assures performance and may further be optimized for performance by utilizing the low-level features of X10 (or C++/Java).

Keywords Transpiler · Fortress · X10 · Abstraction · Xtext · Xtend

The authors' names are listed in alphabetical order.

✉ Anshu S. Anand
anshusanand2001@gmail.com

Karthik Sayani
kartik.sayani.3196@gmail.com

R. K. Shyamasundar
shyamasundar@gmail.com

[1] Indian Institute of Information Technology Allahabad, Prayagraj, India

[2] Indian Institute of Information Technology Vadodara, Gandhinagar, India

[3] Indian Institute of Technology Bombay, Mumbai, India

1 Introduction

The accelerating pace of advances in computational science has challenged the scientific community to use advanced computing capabilities to understand and solve complex problems from various scientific disciplines like numerical simulations, model fitting, and data analysis. These problems involve large computations that are usually executed on multi-node computing clusters with each node comprised of multi-core processors and accelerators that specialize in number crunching and parallel computing. To exploit the available hardware, several high-performance languages have been developed, which enable programmers to run an application on thousands of threads over hundreds of computing units. For an experienced computer programmer, it is a matter of a few days to learn and get familiar with the features of these languages. However, all these languages have their own sophisticated syntaxes and constructs. This leads to two challenges:

- Firstly, there is much boilerplate code involved in writing a parallel program. Programmers themselves have to spawn and manage the thread pool and are also responsible for synchronization and joining of these threads. For example, for a simple FOR loop construct, the body of which is required to be executed in parallel, a programmer has to invoke a certain number of threads, write code to issue tasks to each thread, and also write separate code for invoking these threads on several processors. The situation gets further complicated when atomic blocks and several computing devices with multiple cores are involved.
- Secondly, for a person not familiar with the syntax of the language, it becomes increasingly difficult to understand how the problem is being solved by a computer. This is important because the end-users of the output of the solutions to these problems are they themselves. It is therefore required that there be some way to get the syntax as close as possible to mathematical notations, which are universally understood.

With these requirements in mind, Guy Steele and his team at Sun Microsystems Inc. began working on Project Fortress in 2002. Fortress [22] was developed as a part of the High Productivity Computing Systems (HPCS) program, along with Chapel [6] and X10 [7, 10], for development of computing systems with a focus on productivity and performance. However, even as Chapel and X10 have evolved as full-fledged programming languages, the development of Fortress language, unfortunately, ceased in the year 2012 due to various reasons. However, Fortress did introduce some exciting syntax and constructs. The goal of the developers was to build a language that was as close as possible to mathematical notation, was efficient, scalable, and at the same time, was as simple as possible to facilitate productivity in high-performance software development.

2 Motivation

The aim of this project is fueled by and carries forward the original goals of Fortress, i.e., to build a high-performance language that is close to mathematical notation, scalable, and simplifies writing parallel code. Since domain/computational scientists are by far the largest user community of HPC, Fortress's proximity to mathematical notation reduces the scope for errors in translating the mathematical equations into programs, while also simplifying debugging, thereby improving their productivity significantly. Thus, providing continued support to such a language becomes absolutely necessary. However, Project Fortress being no longer under development, and in its current state only having a working interpreter, all the features provided by Fortress become unreachable to the entire scientific computing community. In this paper, we try to address this problem using the X10 programming language.

X10 is an object-oriented and statically-typed language that has found wide popularity in the HPC community. Both X10 and Fortress unite in the view of the abstract programming model that they follow: Partitioned Global Addressing Space (PGAS), wherein the global addressing space is logically divided into several partitions such that each partition is local to some processing element (called Places in X10). This approach is particularly helpful in exploiting data locality in multi-core, multi-node clusters.

Since both Fortress and X10 work on very similar execution models and are both compiled and ran on JVM, it opens up an opportunity to build a source-to-source compiler (transcompiler/transpiler) for Fortress code to be converted into X10. Thus, in this paper, we present a transpiler that translates Fortress programs into X10, enabling users to still write programs in the Fortress syntax that is similar to mathematical notation, and at the same time, get better performance using X10's infrastructure. Our work also supports Multiresolution language philosophy [17] that simplifies parallel programming by allowing the domain scientists to use the high-level specification for convenience and productivity, and at the same time, provide fine-grained control to the HPC programmers using low-level representations for performance. Since the X10 compiler has two backends: C++ and Java, X10 itself can be compiled to both C++ and Java, and therefore provides more flexibility to the HPC programmers for fine-grained optimizations of the original Fortress code.

The paper has been organized as follows: In the next section, we give a brief background of the Xtext framework and the Xtend language used to realize the transpiler. The architecture of the translation process from Fortress to X10 is described in detail in Sect. 3.2. We then present a case study of Buffon's Needle algorithm and give its Fortress implementation and the translated X10 code in Sect. 4. The results of run-time comparisons of benchmark applications in Fortress and the corresponding translated X10 code is discussed in Sect. 5.

3 Background

The Fortress-to-X10 Transpiler has been built using Xtext [25] which is a framework for development of programming languages especially parser-based external Domain-Specific Languages (DSL) [12, 23]. Unlike internal DSLs which are written inside an existing host language in the form of an API, external DSLs are parsed independently of the host language and have their own syntax. The transpiler also employs the Xtend language [24] for code generation. We first give a brief overview of the Xtext framework and the Xtend language, and then discuss the similarities and differences in Fortress and X10 languages.

3.1 Xtext Framework

Xtext provides complete infrastructure, including parser, linker, type-checker, compiler as well as editing support for Eclipse. It uses the LL(*) parser generator of ANTLR in the background, allowing it to cover a wide range of syntax.

In order to build a DSL, Xtext requires the DSL's grammar to be defined using the Xtext grammar language. Xtext grammar language itself is an EBNF-like DSL developed using Xtext [3]. Now, to derive the various language components, we need to execute the *Generate Xtext Artifacts* command. This generates the following:

1. a metamodel based on Eclipse Modelling Framework's (EMF) Ecore model [21]. From this Ecore model, a Java-API is generated that allows the AST to be accessed programmatically.
2. an ANTLR-based parser that generates the abstract syntax tree (AST) for textual DSL models.
3. a full-featured text editor with support for code highlighting, syntax coloring, content assist, code navigation, etc.

3.2 Xtend Language

Xtend is a statically typed template language for implementing generators, interpreters, and model transformations. Since each of these require access to AST, Xtend enables programmatic access to it using the Java-API mentioned in Sect. 2.1. Since Xtend is compiled to Java, it seamlessly integrates with all existing Java libraries. Like Xtext's grammar language, Xtend language too has been built using Xtext. It provides a rich set of language features like:

- lambda expressions
- template expressions
- active annotations
- type-based switch statements
- polymorphic method invocation

The meta models of Xtext DSLs are represented as Ecore models and since Xtend itself is built using Xtext, each Xtend program is also represented as an Ecore model. The relationship between Xtext, Xtend and Ecore has been discussed in detail by Klaus Birken [4].

3.3 Fortress and X10 Languages

Here, we introduce Fortress and X10 languages through a comparison of a few important programming aspects:

1. **Programming Model:** As mentioned in Sect. 1, both Fortress and X10 use the PGAS programming model which provides an abstraction of a single shared address space even though the address space is partitioned into regions based on the underlying NUMA architecture. X10, in addition, also supports asynchronous operations and control flow, which permits the creation of asynchronous tasks locally and globally, due to which it is said to be an Asynchronous PGAS (APGAS) language.
2. **Basic Execution Model:** The basic unit of execution in Fortress is a thread - implicit or explicit (launched using `spawn`), while in X10 it is known as an activity—a lightweight thread or a user-level thread that is much cheaper to create and manage than kernel-level threads.
3. **Memory Abstraction:** Unlike X10 and Chapel, which provide flat memory abstractions, Fortress provides a hierarchical abstraction of the target architecture. This is realized using regions that map to an element of the system's hierarchy i.e., node, processor, core, or memory, thereby forming a hierarchical tree. Every thread, object, and array element has an associated region in Fortress, which can be queried using the function *region* provided by Fortress. X10, on the other hand, uses the notion of places that represent various computational units with local memory. Both Fortress and X10 allow computation to be placed near data using the same construct: `at`.

Fortress and X10 have many similar features. While Fortress uses `spawn-at`, `for-spawn`, and `at-atomic`, X10 uses `at-async`, `for-async`, and `at-atomic` for asynchronous remote tasks, nested parallelism, and remote transactions, respectively. Also, X10 provides distributed arrays for data distribution, while Fortress provides arrays, vectors, and matrices, that are assumed to be distributed across the machine [22]. *Tuples* in Fortress are also similar to *Points* in X10.

Fortress, in addition, supports several unique features that were aimed at improving the productivity of programmers. Here, we list a few:

1. Growable syntax [2, 20]
   Growable syntax allows growing a programming language using syntactic abstraction. Thus, the growable syntax of Fortress allows it to adapt to the changing needs of users by providing support for adding new constructs in libraries by defining them in terms of existing constructs.
2. Dimensions and units [1]
   Many applications involve representing physical quantities that are usually

expressed as raw numbers. By providing adequate support for units and dimensions, Fortress eliminates bugs that may arise due to misrepresentation of different physical measurements. For example, addition/subtraction/comparison of quantities that are expressed in different units.

3. Function contracts [22]
   Function contracts allow a user to express certain semantic properties that cannot be expressed through the static type system.
4. High-level combinators [11]
   It allows nested data structures to be generated through a set of primitives, called Generators of Generators (GoGs).

Further description of Fortress language follows in Sect. 3.2.

4 Translation from Fortress to X10

In this section, we describe the challenges in compiling Fortress to X10 and describe the implementation of the Fortress-to-X10 transpiler along with an illustration of translation of a few key constructs.

4.1 Challenges

Targeting X10 for translation has raised many challenges in the design of the transpiler. Issues such as extensive usage of Left Recursion in the original Fortress grammar expressed in Parser Expression Grammar and inclusion of unimplemented features such as Dimensions, Coercion, Tests, and Properties embedded in the Fortress Grammar presented a major task of filtering and adapting the essential abstractions from the Fortress grammar while staying true to the objectives of the project. The resulting modified Fortress Grammar faced the following major challenges:

- **Multiplication Operator**:
  Originally, multiplication in Fortress is implied by juxtaposing two operan-ds together. Juxtaposition itself is an overloaded operator that is given semantic actions at run-time, i.e., When the left operand is a function, number or a string, juxtaposition performs function application, multiplication and string concatenation, respectively.
  Such an operator was possible to be defined in Parsing Expression Grammar (PEG) due to infinite look-ahead, which is not the case with LL(*) parser. And hence, multiplication operator '*' was used.
- **Parsing Nested Expressions**:
  Since Xtext doesn't support left-recursive parser rules, parsing of nested expressions is difficult due to their recursive nature. To get rid of left-recursion, the grammar needs to be left-factored. Operator precedence is handled by defining an order of delegation. An operator with higher precedence has its rule listed above other operators.

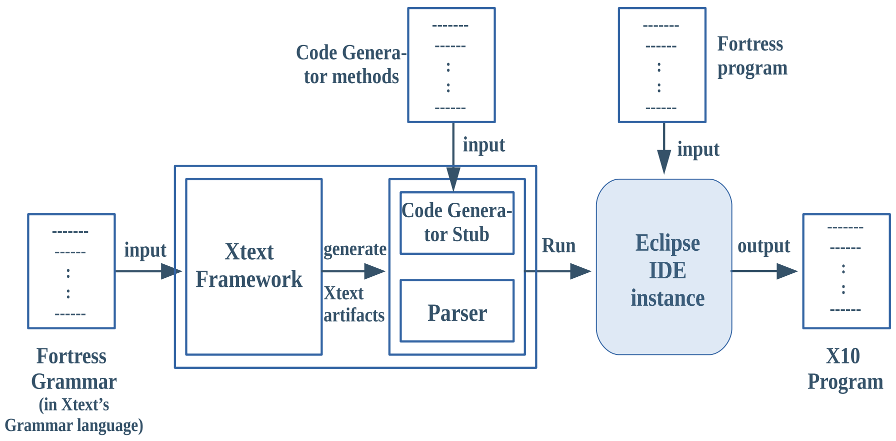


Fig. 1 Architecture of fortress-to-X10 transcompilation

- **Dimensions and Units**:
  Since Fortress is being translated to X10, which doesn't provide support for dimensions and units, providing support for them in Fortress poses a challenge. This is because dimension-checking would require manipulation of dimensions according to a dimensional algebra. There are type systems that model units as types so that dimensional analysis is reduced to type checking [13]. However, due to the complex semantics of units, more powerful algorithms are required to perform type checking for unit correctness. We discuss implementations of programming languages supporting dimensions and units in Sect. 3.5.
- **Return Statements**: Being an expression-oriented language, Fortress has every construct as an expression that returns some value. For example, in the following code:

```
factorial(n:ZZ64):ZZ64 =
    if n === 0
        then 1
    else
        n*factorial(n-1)
end
```

the IF block is an expression that returns a value. We illustrate an example in Sect. 3.3, where we revisit the same code (but in a different context) and show how we address it.

4.2 An Architecture for Translation from Fortress to X10

The Fortress to X10 transpiler, built using the Xtext framework and the Xtend language, takes as input a Fortress program and translates it into the corresponding X10 code. The architecture of Fortress-to-X10 transcompilation is shown in Fig. 1. To implement the Fortress to X10 transpiler, the Xtext project takes as input the grammar of Fortress language (obtained from the Fortress language specification [22]) specified in Xtext's grammar language. On generating the Xtext artifacts for the Fortress grammar, apart from the parser and other infrastructure being generated, a code generator stub is put into the runtime project. The code generator is then written using the Xtend language by defining methods to translate each element of the EMF's metamodel. On executing the project, a new instance of Eclipse IDE is generated with support for functionalities like code completion, syntax highlighting, syntactic validation, linking errors, the outline view, find references, etc. This enables complete Eclipse support for programs written in the Fortress language. To translate Fortress programs to X10, the code generator invokes (Xtend) methods corresponding to each element of AST generated by the parser. Since the DSL Xtext editor is already integrated in the automatic building infrastructure of Eclipse, the generator will be automatically called when the source is written/modified in our DSL (Fortress, in our case). Thus this change is automatically reflected in the translated code. We now illustrate a simple application implementing a threshold function, which is translated from Fortress to X10. Listing 1 shows the application written in Fortress.

```
static component threshold
static export Executable
  threshold(x: ZZ32): ZZ32 =
    if x>127
        then 1
    else
        0
  end
end
```

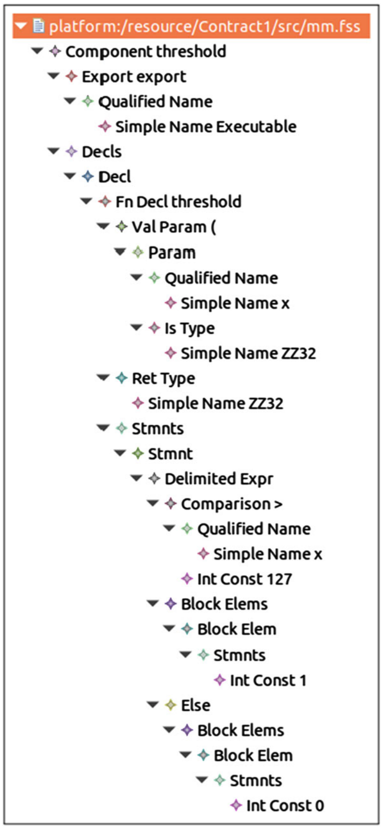
Listing 1 Fortress code implementing a threshold function

```
import x10.io.Console;
import x10.lang.Math;
import x10.array.Array_1;
import x10.array.Array_2;
import x10.array.Array_3;
import x10.util.Random;

public class threshold{

  static def threshold(x:Int):
    Int{
      var result:Int;
    if((x > 127)){
          result = 1;
    }
      else{
          result = 0;
      }
      return result;
  }
}
```

Listing 2 Translated X10 code



```
▼ platform:/resource/Contract1/src/mm.fss
  ▼ Component threshold
    ▼ Export export
      ▼ Qualified Name
          Simple Name Executable
    ▼ Decls
      ▼ Decl
        ▼ Fn Decl threshold
          ▼ Val Param (
            ▼ Param
              ▼ Qualified Name
                  Simple Name x
              ▼ Is Type
                  Simple Name ZZ32
          ▼ Ret Type
              Simple Name ZZ32
          ▼ Stmnts
            ▼ Stmnt
              ▼ Delimited Expr
                ▼ Comparison >
                  ▼ Qualified Name
                      Simple Name x
                    Int Const 127
                ▼ Block Elems
                  ▼ Block Elem
                    ▼ Stmnts
                        Int Const 1
                ▼ Else
                  ▼ Block Elems
                    ▼ Block Elem
                      ▼ Stmnts
                          Int Const 0
```

The Figure to the right of Listing 1 shows a representation of the AST for the application, obtained using the *Sample Reflective Ecore Model Editor* tool. The translated X10 code is shown in Listing 2.

4.3 Illustration of Translation of a Few Key Constructs

For illustration, we show the translation of a few key constructs in Fortress to X10.

4.3.1 Translation of If-Then Block

Listings 3, 4 show a Fortress code snippet of an *if-then* block and the corresponding translated X10 code, respectively.

```
if 0.0 < val AND val < 1.0 then
    println("Hello")
end
```

Listing 3 Fortress code of `if-then` block

```
if(((0.0f < val)&&(val < 1.0f))){
    Console.OUT.println("Hello");
}
```

Listing 4 Translated X10 code

The Xtend method generates the if-then block in X10 syntax using simple string operations. As with Java, the string literals in Xtend (enclosed in ”’ ”’) can be concatenated with the ’+’ operator. The transpiler traverses through the AST and translates every element of *if* to the corresponding code in the X10 syntax.

4.3.2 Translation of for Loop

Since *for* loops in Fortress are parallel by default, the equivalent translated X10 code can be implemented using *async* and *finish*. For each loop iteration, a new asynchronous activity in X10 is spawned using *async*. Listings 5, 6 show a code snippet of *for* loop in Fortress and the corresponding translated X10 code, respectively.

```
for i<- 0#1000 do
    a[i] := SQRT((a[i])^2 + (a[i])^3)
end
```

Listing 5 Fortress code of `for loop`

```
finish for(i in 0n..(1000n-1))
    async{
        a(i)=SQRT(((Math.pow(a(i),2n))+(Math.pow(a(i),3n))));
    }
```

Listing 6 Translated X10 code

Parallelism in *for* loops is specified by the generators used. Thus, if there is any dependency that limits parallel execution, the programmer should specify the use of sequential generator (eg. i $\leftarrow seq(1:n)$) that forces the iterations to be performed sequentially.

The grammar rules for *for* loop in Fortress, specified in the Xtext's grammar language is listed below:

```
DelimitedExpr:
'for' gen=Generators dofront=DoFront 'end';

Generators:  binding=Binding
                ( ',' clause+=GenClause)* ;

Binding: idtup=Qualified '<-' g=GenSource
  | idtup=Qualified '<-'
  seq='seq' '('g=GenSource')';

GenSource: Expr({GenSource.start=current}
                            '#' end=Expr)?;

GenClause: binding=Binding
      | expr=Expr ;
```

A *for* loop consists of the 'for' keyword followed by a generator clause list. In Fortress, comma-separated generator clause lists are utilized to express parallel iterations (eg. for i← 1:m, j← 1:n do ..... end). Thus, the body of a 'for' loop is evaluated for every combination of values bound in the generator clause list (i, j in the above example), in parallel. The generator clause list begins with a binding that consists of one or more identifiers followed by the token '←' and a generator source that is essentially a sub-expression that specifies the range of values for which the *for* loop is to be evaluated.

4.3.3 Translation of Function Contracts

Function Contract is a key feature of Fortress that allows a function to impose certain conditions on its execution. They enable us to express semantic properties that cannot be expressed through the static type system.

Fortress allows three optional clauses in the function's declaration that are evaluated in the scope of the function body: `requires`, `ensures`, and an `invariant` clause. A brief description of these clauses is given below:

1. `Requires`
   It specifies constraints (as a sequence of comma-separated boolean expressions)

that the argument to a function must satisfy. The body of the function is evaluated only if these expressions evaluate to true, or else an exception (CallerViolation) is thrown.

2. `Ensures`
   An `ensure` clause is evaluated after a `requires` clause. It consists of a sequence of ensures subclauses, each comprising of a boolean expression, followed by an optional `provided` subclause. The `provided` subclause consists of the keyword `provided` followed by a boolean expression. The boolean expression preceding `provided` is evaluated after the function body is evaluated, only if the expression following `provided` evaluates to true or in the absence of the `provided` subclause. If this expression (preceding `provided`) evaluates to *false*, an exception *CalleeViolation* is thrown.
3. `Invariant`
   It specifies a sequence of expressions (of any type), enclosed by curly braces. These expressions are evaluated both before and after a function call. For each expression e, if the value of e evaluated before the function call is not equal to the value of e evaluated after the function call, an exception *CalleeViolation* is thrown.

```
factorial(n:ZZ64):ZZ64
  requires {n>=0}
  ensures{result >= 0}=
  if n === 0
    then 1
  else
    n*factorial(n-1)
end
```

Listing 7 Fortress code with function contracts

```
static def factorial (n:Long):Long{
     var result:Long;
     if((n >= 0)){
          if((n == 0)){
               result = 1;
          }
          else{
               result = (n*factorial((n-1)));
          }
     }
     else
          throw new Exception("CallerViolation");
     if((result >= 0)== false)
          throw new Exception("CalleeViolation");
     return result;
}
```

Listing 8 Translated X10 code

Listings 7, 8 show an example of the usage of function contracts in Fortress and the corresponding translated code in X10. The code also highlights the ability of Fortress to succinctly express these semantic properties, which could be realized in X10 only with much extra boilerplate code, affecting the readability of code. Also, since the function has a return type (Long), the transpiler declares a variable *result* that captures the values of expressions returned by the `if-then` block, which is eventually returned at the end.

4.4 Objects and Traits

Objects in Fortress are the same as classes in the standard object-oriented languages like Java or C++, and consist of fields and methods in their body. Object declarations do not allow abstract methods. Listings 9, 10 show a Fortress code snippet of an *object* declaration and the corresponding translated X10 code, respectively.

```
object factorial
    fact(i:ZZ64):ZZ64 =
      if i===0
      then 1
    else
      i*fact(i-1)
    end
end
```

Listing 9 Fortress code of an object declaration

```
public class factorial {

    def fact(i:Long):Long{
      var result:Long;
      if((i == 0)){
          result = 1;
      }
      else{
          result = (i*fact((i-1)));
      }
            return result;
    }
}
```

Listing 10 Translated X10 code

Traits Unlike other object-oriented languages, Fortress does not allow objects to be extended by other objects. For instance, Java supports multiple inheritance by augmenting single inheritance with interfaces. However, Fortress still allows multiple inheritance of methods (and not fields) through traits. Traits are very similar to interfaces in Java, but in addition to abstract methods, trait declarations also allow concrete implementations of methods. Thus, Fortress objects can extend multiple traits and inherit (abstract and/or concrete) methods from them. Moreover, traits themselves may extend other traits. Thus, Traits are an effective programming language mechanism to reduce redundancy in code through reuse of methods.

Traits are supported by neither X10, Java, or C++, and further, Fortress itself does not provide interfaces. Therefore, Traits can be supported in Fortress in a similar way by which they are implemented in Scala [19]. This is realized by compiling each Fortress *Trait* declaration into an X10 interface and an X10 helper class. While the X10 interface holds the abstract methods defined in the trait, the concrete implementations themselves are held in the X10 helper class as static methods. The helper class remains invisible to the user and the implementation of static methods can be accessed only through reference to the interface. An example of the expected translation is shown in Listings 11, 12.

```
trait abc
    display1() =  println("implementation of display1 method")
end
object xyz extends abc
    display2() =  println("implementation of display2 method")
end
```

Listing 11 Fortress code with Object and Trait

```
interface abc {
    void display1();
}

abstract class abc$class {
    public static void display1(abc a) {
        Console.OUT.println("This is the implementation of
    display1 method");
    }
}

class Foo implements Trait {
    public void display1() {
        abc$class.bar(this);
    }
    public void display2() {
        Console.OUT.println("This is the implementation of
    display2 method");
    }
}
```

Listing 12 Translated X10 code

Here, the abc trait is compiled into the interface, abc and the abstract class, abc$class. The concrete implementation of the display1 (static) method is accessed by using the reference to the abc interface.

4.5 Units and Dimensions

Following the loss of NASA's Mars Climate Orbiter probe in 1999 due to a bug resulting from mismatched quantities, several attempts have been made to extend programming languages with support for dimensions and units. Although attempts in this direction are known from 1970s, these were limited by the lack of polymorphism and user-definable dimensions [9]. Andrew J. Kennedy overcame these limitations [14] and provided support for dimension checking for the general purpose functional programming language ML. The implementation identified a canonical unit for each dimension, other units of the same dimensions are then represented in this canonical unit using the respective conversion method. There were also other attempts for languages like F# [15] and Haskell [5]. However, all of these were limited to structurally typed functional languages. Allen et al. [1] in 2004, as part of the Fortress project by Sun, introduced support for static checking of units and dimensions for nominally typed object-oriented languages. Their implementation requires units and dimensions to behave like both types and values. To integrate units and dimensions with object-oriented types, each dimension is specified as a class and allows classes to be instances of other classes. Essentially the Java language was extended with *metaclasses*, *instance classes*, and *abelian classes*, of which dimensions are an instance.

Since neither X10, nor C++ and java provide support for units and dimensions, any attempt to provide this support in Fortress has to be provided by the transpiler, and thus any such mismatch of units and error will be reflected by the parser and the compiled X10 program will be void of these units. A possible direction to implement units and dimensions can be through generics, by using units and dimensions as type parameters. All units are transformed to their canonical type of the respective dimension and dimensional analysis is performed using the canonical types. These type parameters are then erased after the type checking, also known as type erasure.

4.6 Tuples

A tuple is a parenthesized, comma-separated series of expressions. Tuples are appropriate when multiple expressions are to be evaluated or passed as arguments, in parallel, and return multiple results. An example of a tuple is (a+b, a-b, a*b). The evaluation of a tuple of expressions,(factorial(a), factorial(b), factorial(c)) could then be translated to X10 using *finish* and *async* statements as shown in Listing 13.

```
finish{
     async factorial(a);
     async factorial(b);
     async factorial(c);
}
```

Listing 13 Translated X10 code

5 Case Study

For the case study, we consider an implementation of Buffon's needle [8] which is a Monte-Carlo Simulation to estimate the value of π. Given a floor with equally spaced parallel lines distance d apart, it finds the probability that a needle of length l lands on any of the lines. This probability is then used to estimate the value of π.

5.1 Fortress Code

The buffon's needle program implemented in Fortress is listed below:

```

component buffons
export Executable
run ( ) : ( ) = do
  needleLength : RR64 = 20.0
  numRows : RR64 = 10.0
  tableHeight : RR64 = needleLength * numRows
  var hits:RR64 = 0.0
  var n:RR64 = 0.0
  start:RR64 = nanoTime()
  println ( "Starting parallel Buffons")
  for i<-1#3000 do
    delta_X = random (2.0) - 1.0
    delta_Y = random (2.0) - 1.0
    rsq = delta_X^2 + delta_Y^2
    if 0.0 < rsq < 1.0 then
      y1 = tableHeight * random (1.0)
      y2 = y1 + needleLength * (delta_Y / SQRT(rsq))
      (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
      temp1 : RR64 = y_L / needleLength
      temp2 : RR64 = y_H / needleLength
      if |/temp1\| = |\temp2/| then
        atomic do hits:=hits + 1.0 end
      end
      atomic do n:=n + 1.0 end
      end
    end
    probability = hits/n
    pi_est = 2.0/ probability
    println("hits =" || hits ||"n = "||n)
    println ("Buffons : estimated Pi= " || pi_est)
    fin = nanoTime( ) - start
    println(fin)
  end
end
```

5.2 The Translated X10 Code

The X10 code of the Buffon's needle program translated using the Fortress-to-X10 transpiler is listed below:

```

import x10.io.Console;
import x10.lang.Math;
import x10.array.Array_1;
import x10.array.Array_2;
import x10.array.Array_3;
import x10.util.Random;
/*needs to import
*/
/*exports
export Executable
*/

public class buffons{

  public static def main(args:Rail[String])
  {
    val needleLength:Double = 20.0f as Double;
    val numRows:Double = 10.0f as Double;
    val tableHeight:Double=(needleLength*numRows)
           as Double;
    var hits:Double = 0.0f as Double;
    var n:Double = 0.0f as Double;
    val start:Double = nanoTime() as Double;
    Console.OUT.println("Starting parallel Buffons");
    finish for(i in 1n..(3000n-1)) async{
      val deltaX = (random(2.0f)-1.0f);
      val deltaY = (random(2.0f)-1.0f);
      val rsq=((Math.pow(deltaX,2n))
       + (Math.pow(deltaY,2n)));
      if(((0.0f < rsq)&&(rsq < 1.0f))){
        val y1 = (tableHeight*random(1.0f));
        val y2 = (y1+(needleLength*(deltaY/SQRT(rsq))));
        val y_L = min(y1,y2);
        val y_H = max(y1,y2);
        val temp1:Double = (y_L/needleLength) as Double;
        val temp2:Double = (y_H/needleLength) as Double;
        if((Math.ceil(temp1) == Math.floor(temp2))){
          atomic{
            hits = (hits+1.0f);
        }
      }
      atomic{
        n = (n+1.0f);
      }
    }
  }
  val probability = (hits/n);
  val pi_est = (2.0f/probability);
  Console.OUT.println("hits =");
  Console.OUT.println(hits);
  Console.OUT.println("n = ");
  Console.OUT.println(n);
  Console.OUT.println("Buffons:estimated Pi= ");
  Console.OUT.println(pi_est);
  val fin = (nanoTime()-start);
  Console.OUT.println(fin);
  }
}
```

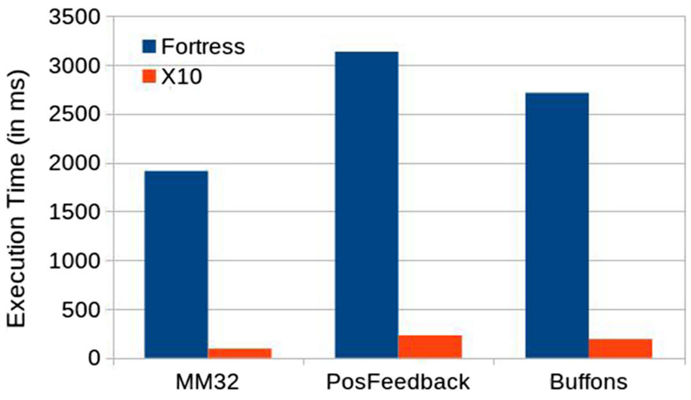


Fig. 2 Fortress versus X10: run-time comparison

The case study demonstrates the translation of Buffon's Needle from Fortress to X10 code using our transpiler framework. It demonstrates the following constructs/ features of Fortress: component, export, run, for loop, and atomic. The codes show the conciseness of the Fortress code and how the syntax is closer to mathematical notation and hence more intuitive. An important distinction between the two languages is the implicit data parallelism in Fortress given by the parallel-by-default for construct.

6 Experimental Results

In this section, we discuss a few benchmark applications in Fortress and their performance with respect to the transformed X10 codes.

All the experiments were conducted on Ubuntu 16.04 LTS system with the Intel i7-4710HQ processor supported by 8 GB of DDR3 RAM. The Fortress codes have been given full JVM memory allowance to create as many implicit threads as needed to achieve the maximum performance. The X10 codes are all run with 4 Places and as many number of activities required by each place. All the run-times are averaged over 5 runs.

We first compare the execution times of the following three applications:

- Matrix Multiplication (*MM32*): employs Divide and Conquer strategy to multiply two matrices of size 32*32.
- Positive Feedback (*PosFeedback*): Given a set of entities, each having a positivity score and a bank of facts, evolving them by supplying random facts.
- Buffon's Pi (*Buffons*) [8]: A Monte Carlo method using Buffon's needle to approximate the value of Pi.

The results of these experiments are shown in Fig. 2.

It is evident from the results that the translated X10 code outperforms the original Fortress code in terms of execution time, with an average speedup of 16x observed across the three experiments. While the speedup is 20.36x for *MM32*, it is 13.63x and 14.13x for *PosFeedback* and *Buffons*, respectively. *MM32* significantly fares better than other applications as here computations are inherently parallel and more numerically intensive and also, no synchronization is needed. *Buffons*, on the other hand, requires at least two atomic operations in each iteration. We also performed experiments with the following benchmark applications:

- Array Stream Benchmarks [16]: It is a synthetic benchmark program that measures memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. The results for array stream benchmarks are shown in Tables 1 and 2.
  In all of the experiments (Int, Int64/Long, and Float), an average speedup of more than 2x was observed for the translated X10 code. Also, in all cases, the highest speedup was observed for the *Triad* operation when compared to other operations.
- NAS Fast Fourier Transform (and Inverse) of 3D grids [18]: This application solves a 3-dimensional partial differential equation using the Fast Fourier Transform. The results for the NAS Fast Fourier Transform (FT) are shown in Tables 3 and 4. For NAS-FT, the translated X10 code clearly outperforms Fortress by a large margin. The highest speedup of 206x is achieved by the X10

Table 1 Fortress stream benchmark test

Int32	OpType	Rate (MB/s)	Avg. time (s)
	Copy	0.76380121485	10.4739294
	Sscale	0.73007764609	10.957738595
	Add	0.91273471055	13.147303221
	Triad	0.80712187147	14.867643195
	Sscale	0.79928385621	10.008959818
Int64	**OpType**	**Rate (MB/s)**	**Avg. time (s)**
	Copy	1.79794160235	8.899065453
	Sscale	1.47913110244	10.81716149
	Add	2.00083967138	11.994964086
	Triad	1.493870698	16.065647472
	Sscale	1.514487715	10.564628446
Float	**OpType**	**Rate (MB/s)**	**Avg. time (s)**
	Copy	1.782398159	8.976669955
	Sscale	1.84008156	8.695266747
	Add	1.756414426	13.664201139
	Triad	2.035071163	11.793199389

Table 2 X10 stream benchmark test

Int32	OpType	Rate (MB/s)	Avg. time (s)
	Copy	8.42345860721409	4.748643267
	Sscale	9.524426067121496	4.199728122
	Add	13.528420739316202	4.435107479
	Triad	8.899139206154164	4.494816754
Long	**OpType**	**Rate (MB/s)**	**Avg. time (s)**
	Copy	16.486417432848583	4.852479341
	Sscale	16.41668567602583	4.873090804
	Add	21.6605982250443	5.540013196
	Triad	14.58383811756954	5.485524411
	Sscale	17.148535071815235	4.66512152
Float	**OpType**	**Rate (MB/s)**	**Avg. time (s)**
	Copy	8.02382936905595	4.985150875
	Sscale	8.483019312473154	4.715302244
	Add	10.898170595394351	5.505511175
	Triad	7.578662969599845	5.277975833

code for Class W. The higher speedup in FT can be explained by the fact that the iterations of FT are independent, and further, more than half of the total time is spent on the core FFT computations and therefore scales well with the number of processors.

We also conducted experiments to comparatively evaluate the performance of the translated X10 code with respect to the X10 code that has been optimized for performance. In this experiment, we observed the execution times of the translated X10 code and the optimized code for *Buffons* and *PosFeedback* for different number of iterations, and computed the speedup achieved by the optimized code w.r.to the original X10 code. The results for *PosFeedback* are shown in Fig. 3.

The results demonstrate the efficacy of the optimized X10 code, with a maximum of 2.5x speedup achieved for $4 * 10^6$ iterations. The result also shows that the speedup increases with the increase in the number of iterations till $4 * 10^6$, and decreases thereafter due to communication overheads. This experiment also demonstrates the Multi-resolution Language philosophy, where the domain scientists could express their problems in Fortress, and the translated X10 code could then be optimized by the HPC programmers.

Table 3 Fortress NAS FT benchmark

Class	Grid	Iterations	Avg. run time (s)
Class T:	2 x 4 x 4	1	0.650441406
Class S:	64 x 64 x 64	1	503.49956373
Class W:	128 x 128 x 32	1	1040.398577308

Table 4 X10 NAS FT benchmark

Class	Grid	Iterations	Avg. run time (s)
Class T:	2 x 4 x 4	1	0.019727665
Class S:	64 x 64 x 64	6	2.653876066
Class W:	128 x 128 x 32	6	5.042912789

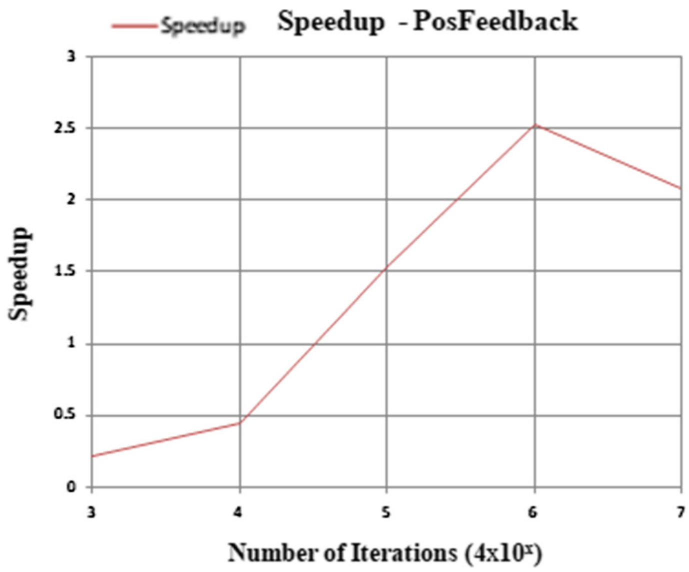


Fig. 3 Fortress versus X10: run-time comparison

7 Discussion

The experimental results show an impressive performance of the translated X10 code with respect to the original Fortress code. The huge performance improvement was as expected because Fortress was an experimental and interpreted language, with an inefficient interpreter. However, by providing this tool for translating Fortress code to X10, we provide an opportunity for better engagement of domain scientists with the X10 community. It will enable the domain scientists to express their problems in a representation that is closer to the mathematical notation, thereby reducing the scope for errors and simplifying debugging. It will also significantly improve their productivity since they can get their implementation to work without worrying about low-level performance aspects. The 90/10 rule [17], which states that almost 90% of a program's execution time is spent within 10% of its code, also suggests better productivity for domain scientists by limiting their focus to problem-solving than on performance improvement. Tuning the program for performance, which requires a good understanding of low-level implementation details, could be realized by HPC programmers. As an example, the domain scientists may write an application using the Fortress arrays. The translated X10 code can then be optimized for performance by taking advantage of the rich support for arrays provided by X10. For example, while *Region* array in X10 could be used as the default choice due to the flexibility they offer, the array implementation could be changed to *Rail* for better performance in certain cases.*Rail* is an intrinsic one-dimensional fixed-size array that supports only up to three dimensions using row-major ordering, thus allowing efficient optimizations on indexing operations. Further, since X10 itself can be compiled to either C++ or Java, HPC programmers have more flexibility to optimize the original Fortress code in a language of their choice.

8 Conclusion

In order to leverage the novel syntactic features of Fortress, we have built a Fortress to X10 transpiler that enables a program written in Fortress syntax to be automatically translated into the corresponding X10 code. We have showcased the tremendous performance gain of X10 codes over the existing Fortress system. Thus, we have shown how the reduction in boilerplate code, readability achieved by Fortress and the impressive performance of X10 can be amalgamated together to form a powerful language for high performance and scientific computing. This also supports the Multiresolution language philosophy that will enable the domain scientists to write programs easily in the Fortress syntax that is close to the mathematical notation, without bothering about performance. The translated X10 code inherently improves performance and can further be optimized for performance by utilizing the low-level features of X10, Java or C++.

There are certain features from the original Fortress implementation that have not yet been implemented in the transpiler like growable syntax and high-level combinators [11]. The current transpiler only supports a single component i.e. a single class and also does not provide support for APIs, which is another name for Interfaces in Fortress. This is the immediate future goal of the transpiler, to provide the notion of multiple classes, APIs, and packages. There are several novel features of Fortress that are not present in the original implementation either, such as dimensions, units, and where clauses. Including these features will further simplify the language for a programmer and hence increase productivity. Another possible addition for the transpiler would be operator overloading and user-defined types and operators. This is an interesting addition that can be useful in various scenarios.

References

1. Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., Steele, G.L.: Object-oriented units of measurement. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), pp. 384–403. ACM, New York (2004)
2. Allen, E., Culpepper, R., Nielsen, J.D., Rafkind, J., Ryu, S.: Growing a syntax. In: Proceedings of Workshop on Foundations of Object-Oriented Languages (2009)
3. Bettini, L.: Implementing DSL with Xtext and Xtend, 2nd edn. Packt Publishing, New York (2016)
4. Birken, K.: Building code generators for DSLs using a partial evaluator for the Xtend Language. In: Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Technologies for Mastering Change, Vol. 8802, pp. 407–424. Springer, New York (2014)
5. Buckwalter, B.: Dimensional: statically checked physical dimensions for Haskell (2008)
6. Callahan, D., Chamberlain, B.L., Zima, H.P.: The cascade high productivity language. In: 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), pp. 52–60. IEEE Computer Society (2004)
7. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. **40**, 519–538 (2005)
8. de Buffon, Georges-Louis Leclerc Comte: Essai da'rithme'tique morale, Histoire naturelle generale et particuliere, Supplement, 4, 685 (1777)
9. Dunfield, J.: A unified system of type refinements, Ph.d thesis, Carnegie Mellon University (2007)
10. Ebcioglu, K., Saraswat, V., Sarkar, V.: X10: Programming for hierarchical parallelism and non-uniform data access. In: International Workshop on Language Runtimes OOPSLA (2004)
11. Emoto, K., Zhenjiang, H., Kakehi, K., Matsuzaki, K., Takeichi, M.: Generators-of-generators library with optimization capabilities in fortress. Euro-Par **2**, 26–37 (2010). https://doi.org/10.1007/978-3-642-15291-7_4
12. Fowler, M.: Domain-Specific Languages. Addison-Wesley, London (2010)
13. Jiang, L., Su, Z.: Osprey: a practical type system for validating dimensional unit correctness of C programs. In: 28th International Conference on Software Engineering (ICSE '06), pp. 262–271. ACM, New York (2006)
14. Kennedy, A.: Programming Languages and Dimensions. Ph.d Thesis, St Catharines College (1995)
15. Kennedy, A.: Types for units-of-measure: theory and practice. In: Central European Functional Programming School, pp. 268–305. Springer, Berlin, Heidelberg (2009)
16. McCalpin, J.D.: STREAM Benchmark. http://www.cs.virginia.edu/stream/
17. Multiresolution Languages for Portable yet Efficient Parallel Programming, Bradford L. Chamberlain, whitepaper (2007)
18. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html
19. Nurkiewicz, T.: Scala traits implementation and interoperability (2013)
20. Ryu, S.: Parsing fortress syntax, In: 7th International Conference on Principles and Practice of Programming in Java (PPPJ '09), pp. 76–84. ACM, New York (2009)
21. Steinberg, D., et al.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, ISBN: 0321331885 (2009)
22. Sun MicroSystems Inc.: The fortress language specification (2008)
23. Voelter, M., et al.: DSL Engineering-Designing. Implementing and Using Domain-Specific Languages, 1–558 dslbook.org, ISBN: 978-1-4812-1858-0 (2013)
24. Xtend Language homepage. http://www.xtend-lang.org
25. Xtext homepage. http://www.eclipse.org/Xtext/