LMSTruffle

combining staging with self-optimizing AST interpretation

Roman Tsegelskyi
CS590

https://github.com/RomanTsegelskyi/lms-truffle

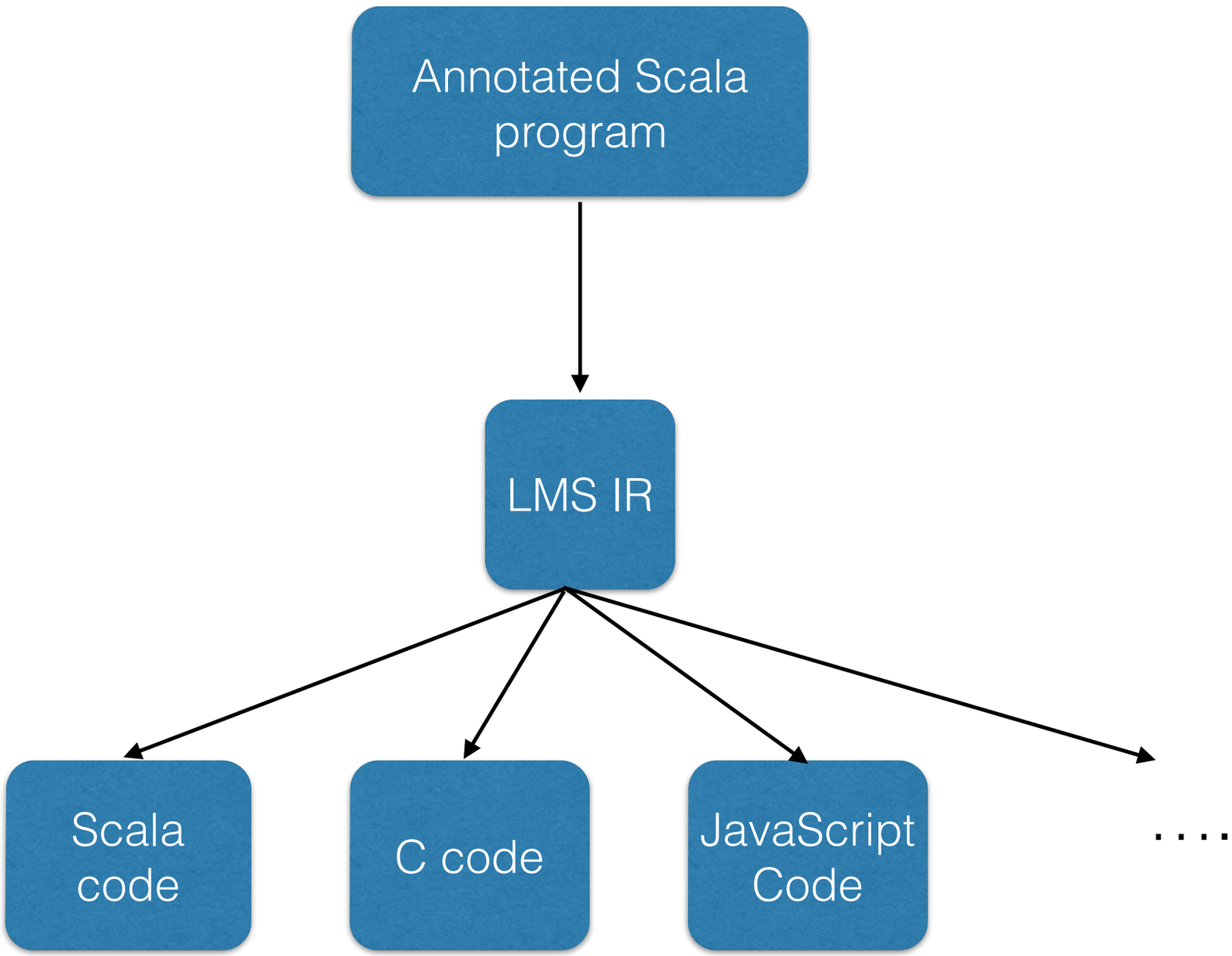
Truffle: A Self-Optimizing Runtime System

- A novel framework for implementing managed languages in Java
- Framework that allows tree rewriting during AST interpretation
- Tree rewrites incorporate type feedback and other profiling information into the tree, thus specializing the tree and augmenting it with run-time information
- The partial evaluation is done by Graal, the just-in-time compiler of our Java VM

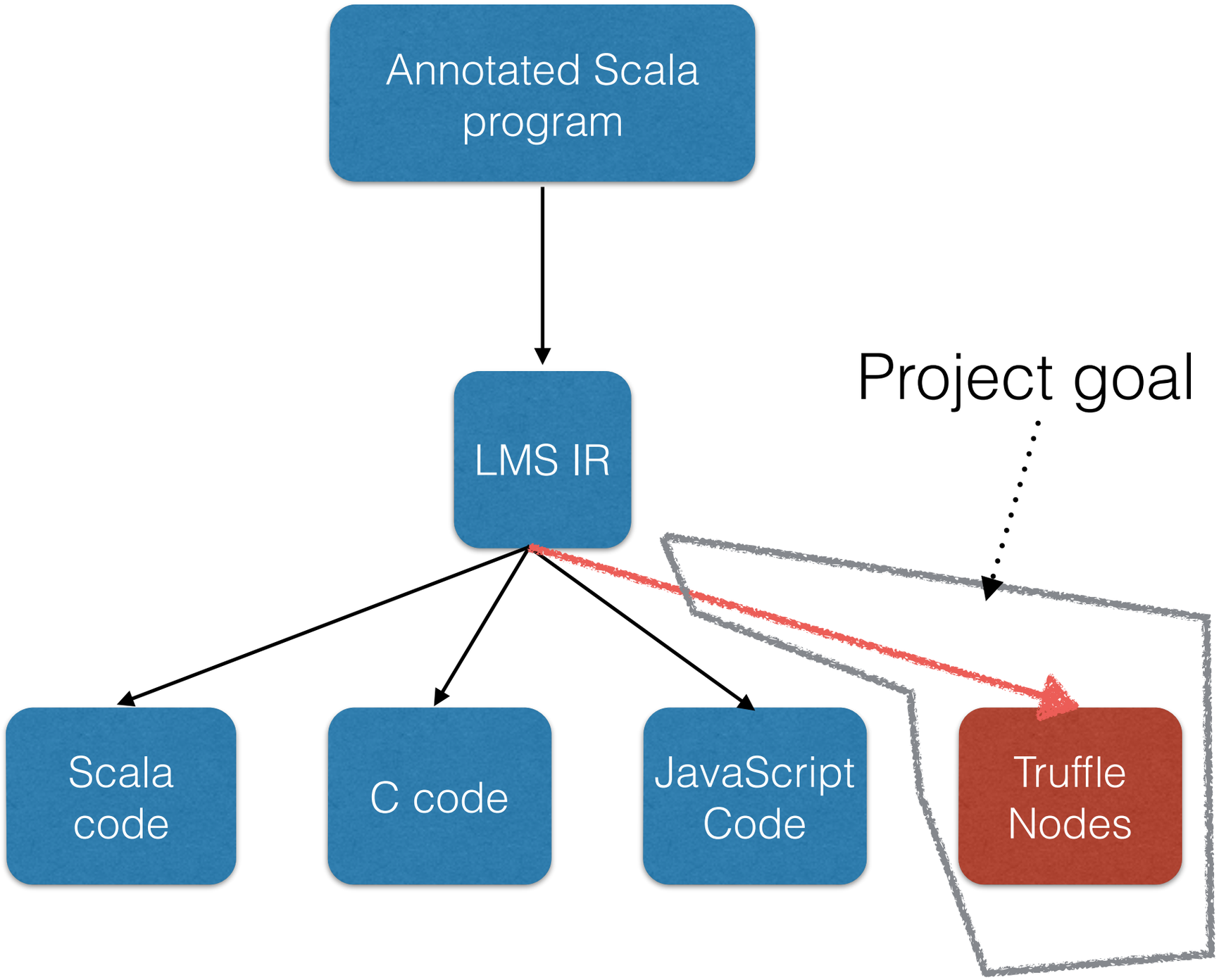
Lightweight Modular Staging (LMS)

- A runtime code generation framework
- Based on multi-stage programming approach
  - "delay" expressions to a generated stage
  - "run" delayed expressions
  - staged program fragments as first class values
- Encode staging information in the types:
  - T means "execute now"
  - Rep[T] means "generate code to execute later"

LMS code generation pipeline



LMS code generation pipeline



Project goals

- Provide a machinery for generating Truffle Nodes
- Explore the benefits of using Truffle nodes as LMS target
- Evaluation and comparison on common use cases of LMS
    1. Power Function
    2. Fast Fourier Transform
    3. SQL interpreter

Power

- Poster child of staging (was already implemented by Prof. Rompf)

```
def power(b: Double, x: Int): Double =
    if (x == 0) 1.0 else b * power(b, x - 1)
```

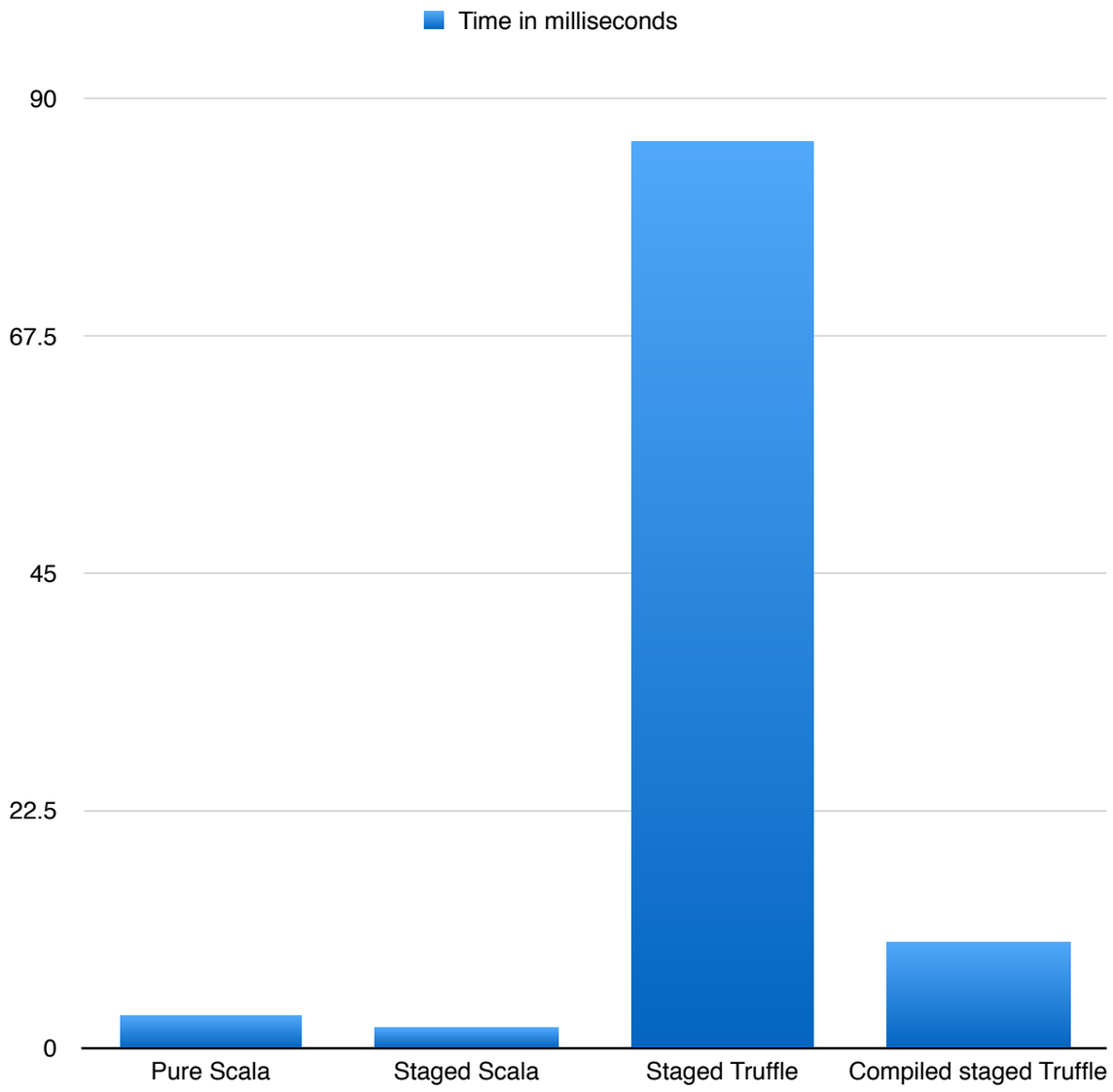
- Scala & Truffle generated code

```
class Power6 extends ((Double)=>(Double)) {
    def apply(x0:Double): Double = {
        val x1 = x0 * x0
        val x2 = x1 * x1
        val x3 = x2 * x2
        val x4 = x3 * x3
        val x5 = x4 * x4
        x5
    }
}
```

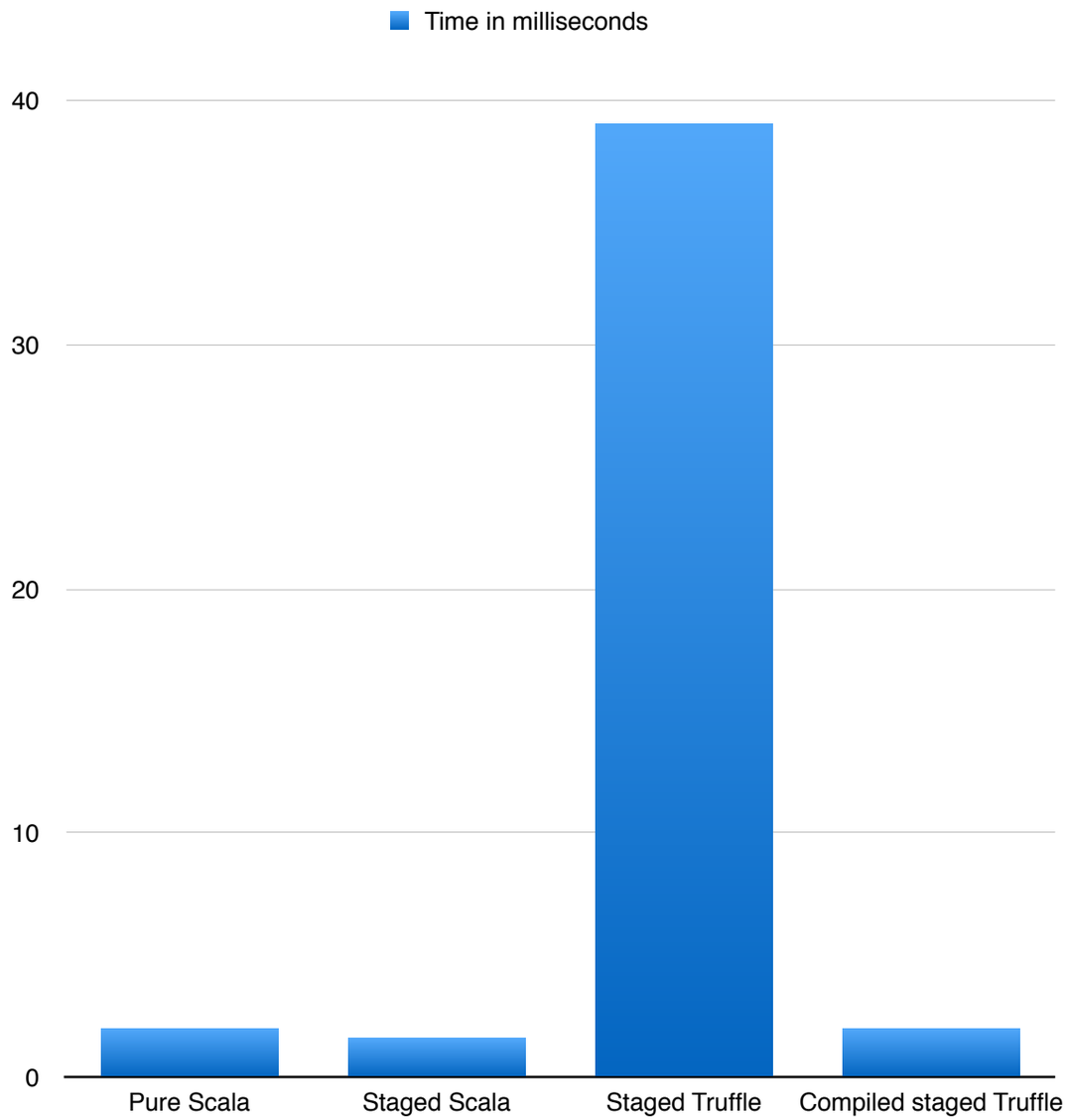
```
Assign([0,x0,Int],GetArg(0))
Assign([1,x1,Int],IntTimes(Sym([0,x0,Int]),Sym([0,x0,Int])))
Assign([2,x2,Int],IntTimes(Sym([0,x0,Int]),Sym([1,x1,Int])))
Assign([3,x3,Int],IntTimes(Sym([0,x0,Int]),Sym([2,x2,Int])))
Assign([4,x4,Int],IntTimes(Sym([0,x0,Int]),Sym([3,x3,Int])))
Assign([5,x5,Int],IntTimes(Sym([0,x0,Int]),Sym([4,x4,Int])))
```

Power

- 1000 calls to power(2, 2048)



- 10000 calls to power(2, 8)



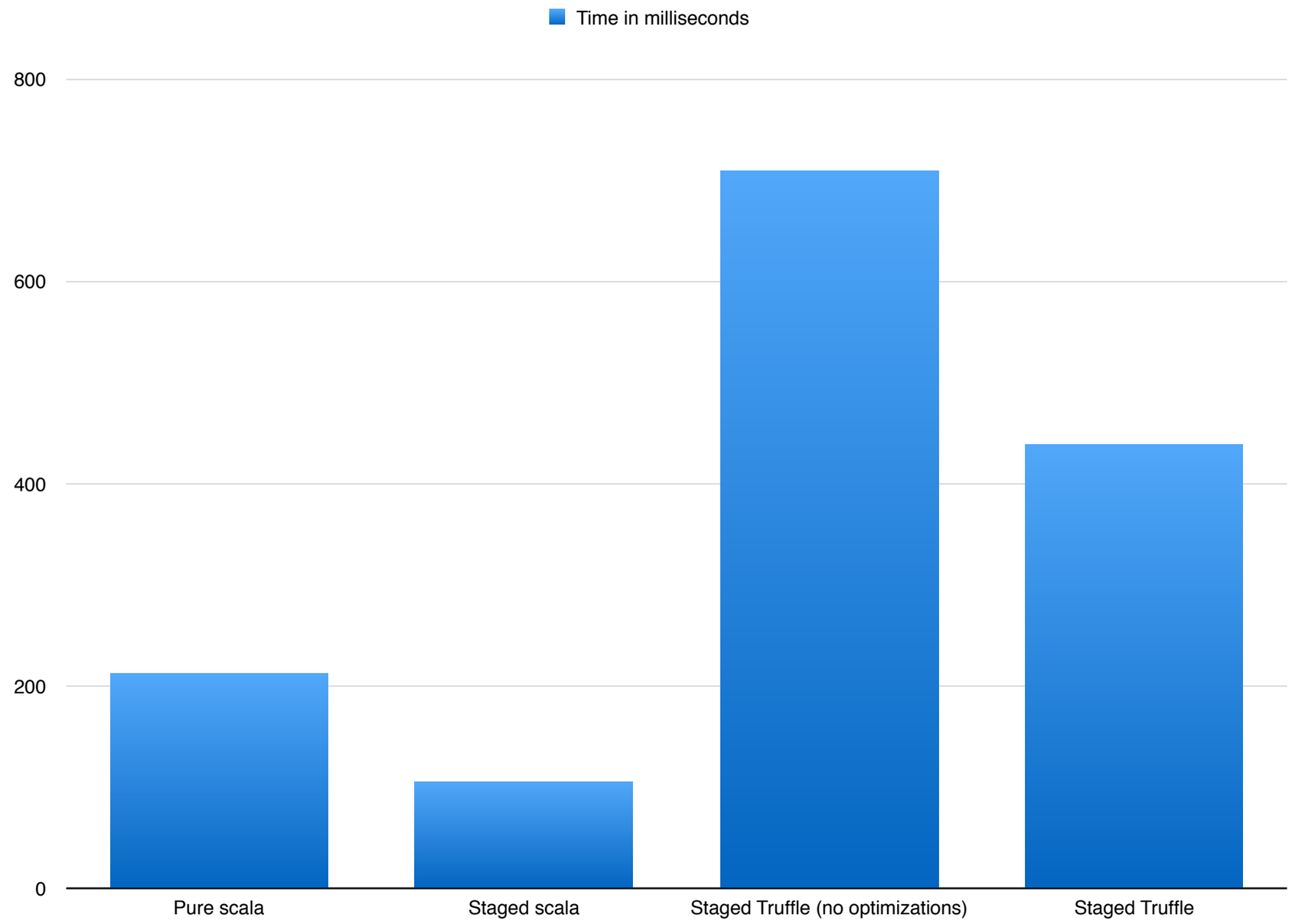
Fast Fourier Transform

- Staging of code for fixed sized arrays
- Required addition of 14 Truffle node times
- Domain specific optimization implemented

```
…
Assign([12,x12,Double],DoubleMinus(Sym([2,x2,Double]),Sym([6,x6,Double])))
Assign([13,x13,Double],DoubleMinus(Sym([3,x3,Double]),Sym([7,x7,Double])))
Assign([14,x14,Double],DoublePlus(Sym([4,x4,Double]),Sym([8,x8,Double])))
Assign([15,x15,Double],DoublePlus(Sym([5,x5,Double]),Sym([9,x9,Double])))
Assign([16,x16,Double],DoubleMinus(Sym([4,x4,Double]),Sym([8,x8,Double])))
Assign([17,x17,Double],DoubleMinus(Sym([5,x5,Double]),Sym([9,x9,Double])))
Assign([18,x18,Double],DoublePlus(Sym([10,x10,Double]),Sym([14,x14,Double])))
Assign([19,x19,Double],DoublePlus(Sym([11,x11,Double]),Sym([15,x15,Double])))
Assign([20,x20,Double],DoubleMinus(Sym([10,x10,Double]),Sym([14,x14,Double])))
Assign([21,x21,Double],DoubleMinus(Sym([11,x11,Double]),Sym([15,x15,Double])))
Assign([22,x22,Double],DoubleMinus(Const(0.0),Sym([17,x17,Double])))
Assign([23,x23,Double],DoubleMinus(Const(0.0),Sym([22,x22,Double])))
Assign([24,x24,Double],DoubleMinus(Const(0.0),Sym([16,x16,Double]))
…
```

Fast Fourier Transform

- 10000 calls to FFT with array of size 8



SQL Interpretation

- Small SQL processing engine
- Produces specialized query. Essentially a query complier

Some csv file

```
Name,Value,Flag
A,7,no
B,2,yes
….
```

Query

```
select Name from t.csv where Flag='yes'
```

```
  def execOp(o: Operator): OperatorNode = o match {
    case Scan(filename, schema, fieldDelimiter, externalSchema) =>
      new ProcessCSVNode(filename, schema, fieldDelimiter, externalSchema)
    case Project(newSchema, parentSchema, parent) =>
      new ProjectNode(newSchema, parentSchema, execOp(parent))
    case Filter(pred, parent) =>
      new FilterNode(pred, execOp(parent))
```

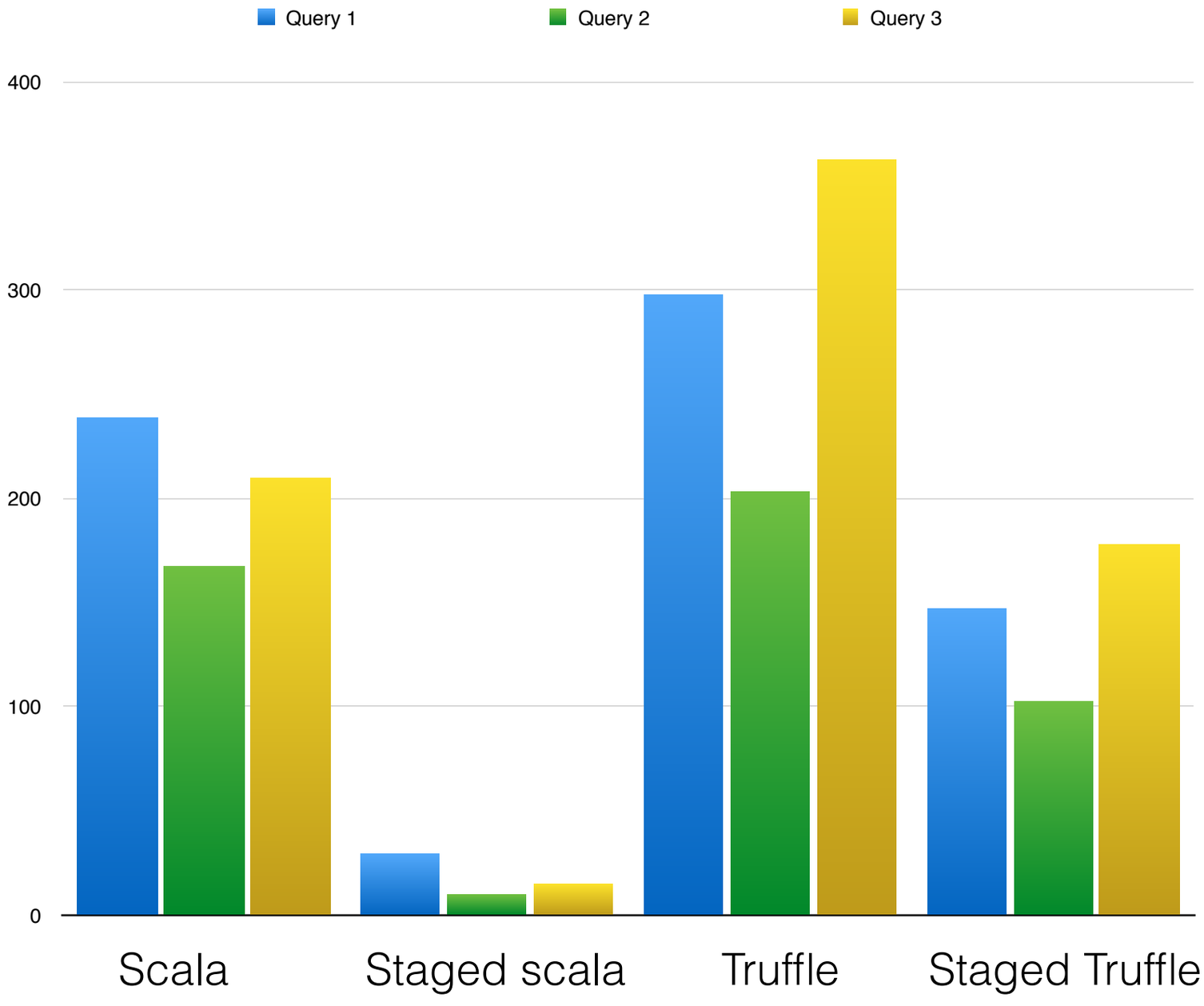
SQL Interpretation

```
class Snippet extends ((java.lang.String)=>(Unit)) {
  def apply(x0:java.lang.String): Unit = {
    val x1 = println("Name")
    val x2 = new scala.lms.tutorial.Scanner("src/data/t.csv")
    val x3 = x2.next(',')
    val x4 = x2.next(',')
    val x5 = x2.next('\n')
    val x16 = while ({val x6 = x2.hasNext
      x6}) {
      val x8 = x2.next(',')
      val x9 = x2.next(',')
      val x10 = x2.next('\n')
      val x11 = x10 == "yes"
      val x14 = if (x11) {
        val x12 = printf("%s\n",x8)
        x12
      } else {
        ()
      }
      x14
    }
    val x17 = x2.close
    ()
}
```

SQL Interpretation

```
Assign([0,x0,Object],GetArg(0))
Assign([1,x1,Object],ScannerNew(Const(src/data/t.csv)))
Assign([2,x2,Object],ScannerNext(Sym([1,x1,Object]),,))
Assign([3,x3,Object],ScannerNext(Sym([1,x1,Object]),,))
Assign([4,x4,Object],ScannerNext(Sym([1,x1,Object]),
))
Assign([11,x11,Object],
  WhileLoop(ScannerHasNext(Sym([1,x1,Object])),
    Assign([5,x5,Object],ScannerNext(Sym([1,x1,Object]),,))
    Assign([6,x6,Object],ScannerNext(Sym([1,x1,Object]),,))
    Assign([7,x7,Object],ScannerNext(Sym([1,x1,Object]),))
    Assign([8,x8,Boolean],StringEq(Sym([7,x7,Object]),Const(yes)))
    Assign([10,x10,Object],
      IfElse(Sym([8,x8,Boolean]),
      Assign([9,x9,Object],PrintFields(Vector(Sym([5,x5,Object])))),
        ))))
Assign([12,x12,Object],ScannerClose(Sym([1,x1,Object])))
```

SQL Interpretation



Future work

- Completely integrate into LMS as target
- Explore more Truffle/Graal optimization possibilities
- Implement HashData structures

What was done

- Implemented Truffle node generation completely for FFT, almost completely for SQL interpretation (without HashJoin and Group)
- Evaluations comparing Scala vs Truffle, JVM vs Graal compiler, etc.