# ZipPy: A Simple Python 3 for the JVM

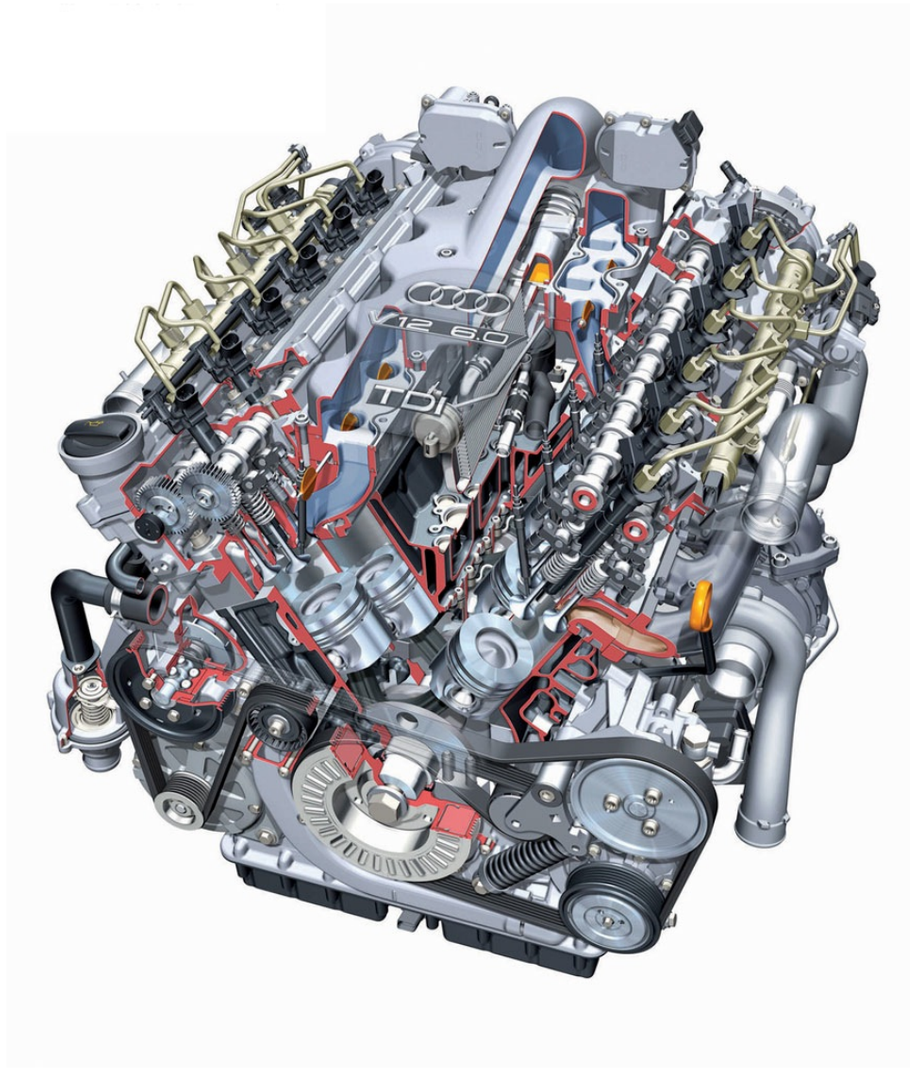
Wei Zhang
UC Irvine

# Python?

- Dynamic languages are here to stay
- People use it too: NumPy, Django
- Concise syntax, good readability
- Py3k is the future

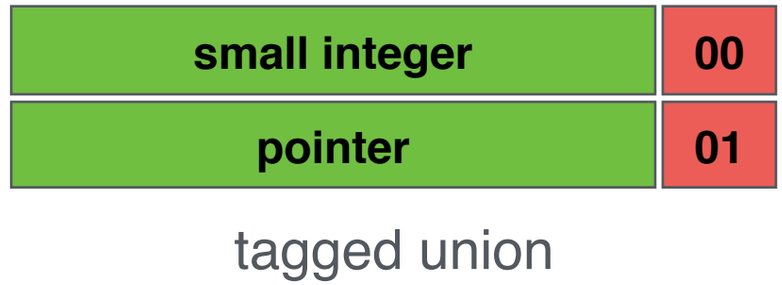
# JVM, the Platform

- 100+ languages
- Robust Memory Management
- Concurrency support
- Development Productivity
- Cross Platform



# JVM, the Challenges

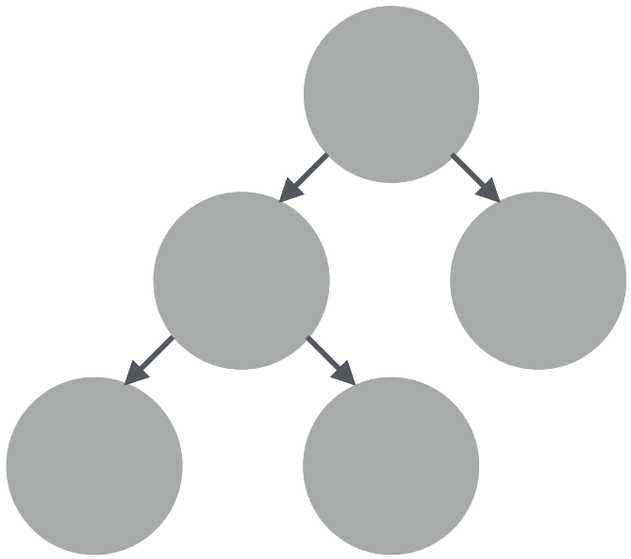
- Value representation
  - ~~Tagged union~~
  - Boxing for numerics
- Runtime type specialization



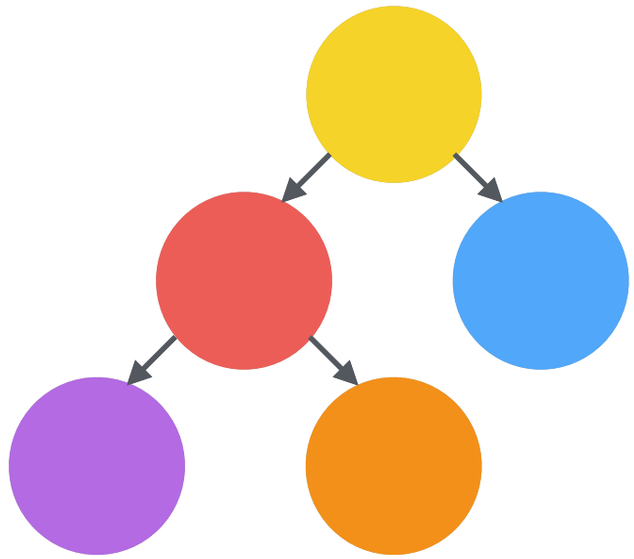

tagged union


all values in heap

# Truffle Framework



Python AST Interpreter
with **uninitialized** nodes



Python AST Interpreter
with **rewritten** nodes

Images from [2]

[1]: S. Brunthaler, Inline Caching Meets Quickening, ECOOP 2010
[2]: T. Würthinger et al., Self-Optimizing AST Interpreters DLS 2012

# Example Python Function

```
def sumitup(n):
  total = 0
for i in range(n):
  total = total + i
return total
```

# AST After Parsing

```
FunctionRootNode
  parameters = ParametersOfSizeOneNode
    parameter = WriteLocalUninitializedNode
      rightNode = ReadArgumentNode
  body = BlockNode
    statements[0] = WriteLocalUninitializedNode
      rightNode = IntegerLiteralNode
    statements[1] = ForWithLocalTargetUninitializedNode
      body = BlockNode
        statements[0] = WriteLocalUninitializedNode
          rightNode = AddUninitializedNode
            leftNode = ReadLocalUninitializedNode
            rightNode = ReadLocalUninitializedNode
      target = WriteLocalUninitializedNode
      iterator = UninitializedCallFunctionNode
        callee = ReadGlobalScopeNode
          load = UninitializedLoadAttributeNode
            primary = ObjectLiteralNode
        arguments[0] = ReadLocalUninitializedNode
    statements[2] = FrameReturnNode
      right = WriteLocalUninitializedNode
        rightNode = ReadLocalUninitializedNode
  returnValue = ReadLocalUninitializedNode
```

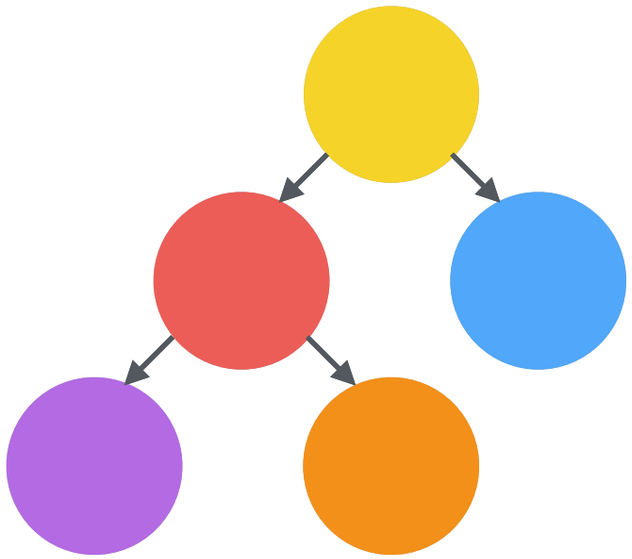
# Specialized AST

```
FunctionRootNode
  parameters = ParametersOfSizeOneNode
    parameter = WriteLocalIntNode
      rightNode = ReadArgumentNode
  body = BlockNode
    statements[0] = WriteLocalIntNode
      rightNode = IntegerLiteralNode
    statements[1] = ForWithLocalTargetPRangeNode
      body = BlockNode
        statements[0] = WriteLocalIntNode
          rightNode = AddIntNode
            leftNode = ReadLocalIntNode
            rightNode = ReadLocalIntNode
      target = WriteLocalIntNode
      iterator = CallBuiltInFunctionDefaultNode
        arguments[0] = ReadLocalIntNode
    statements[2] = FrameReturnNode
      right = WriteLocalIntNode
        rightNode = ReadLocalIntNode
returnValue = ReadLocalIntNode
```

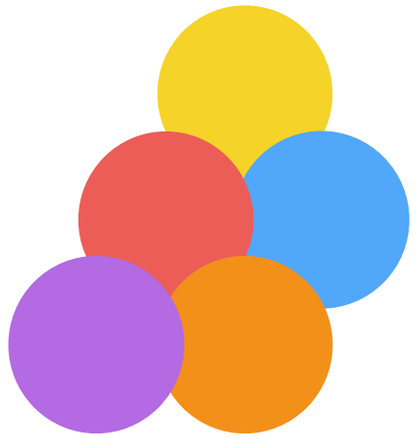
# AddNode

```
@NodeChildren({
  @NodeChild(value = "leftNode", type = PNode.class),
  @NodeChild(value = "rightNode", type = PNode.class)})
public abstract static class AddNode extends PNode{

  @Specialization(rewriteOn = ArithmeticException.class, order = 0)
  int doInteger(int left, int right) {
    return ExactMath.addExact(left, right);
  }

  @Specialization(order = 1)
  BigInteger doBigInteger(BigInteger left, BigInteger right) {
    return left.add(right);
  }
[...]
  @Generic
  Object doGeneric(Object left, Object right) {
    throw Py.TypeError("unsupported operand type(s) for +:");
  }
}
```

# Partial Evaluation



Python AST Interpreter
with **rewritten** nodes



**Compiled** Python program

Images from [1]

[1]: T. Würthinger et al., One VM to Rule Them All Onward! 2013

# Example Python Function Specialized

```
int sumItUp(int n) {
  int total = 0;

  for (int i = 0; i < n; i++) {
    total = total + i;
  }

  return total;
}
```

# Machine Code for the Loop

```
            jmp L7

L6:     mov         ecx, edx
        add         ecx, ebp
        jo          L8
        mov         edx, ebp
        incl        edx
        mov         esi, ebp
        mov         ebp, edx
        mov         edx, ecx
L7:     cmp         eax, ebp
        jle         L9
        jmp         L6
L8:     call        deoptimize()

L9:
```

# Performance of Our Example

```
def sumitup(n):
  total = 0
  for i in range(n):
    total = total + i
  return total
```

50,000 invocations of sumitup(50,000)

| | |
|---|---|
| CPython 2.7 | 110 sec. |
| CPython 3.3 | 147 sec. |
| PyPy 2.1 | 4.0 sec. |
| ZipPy | 3.8 sec. |

Peak performance after warmup runs, so that method is compiled

# Call Graph

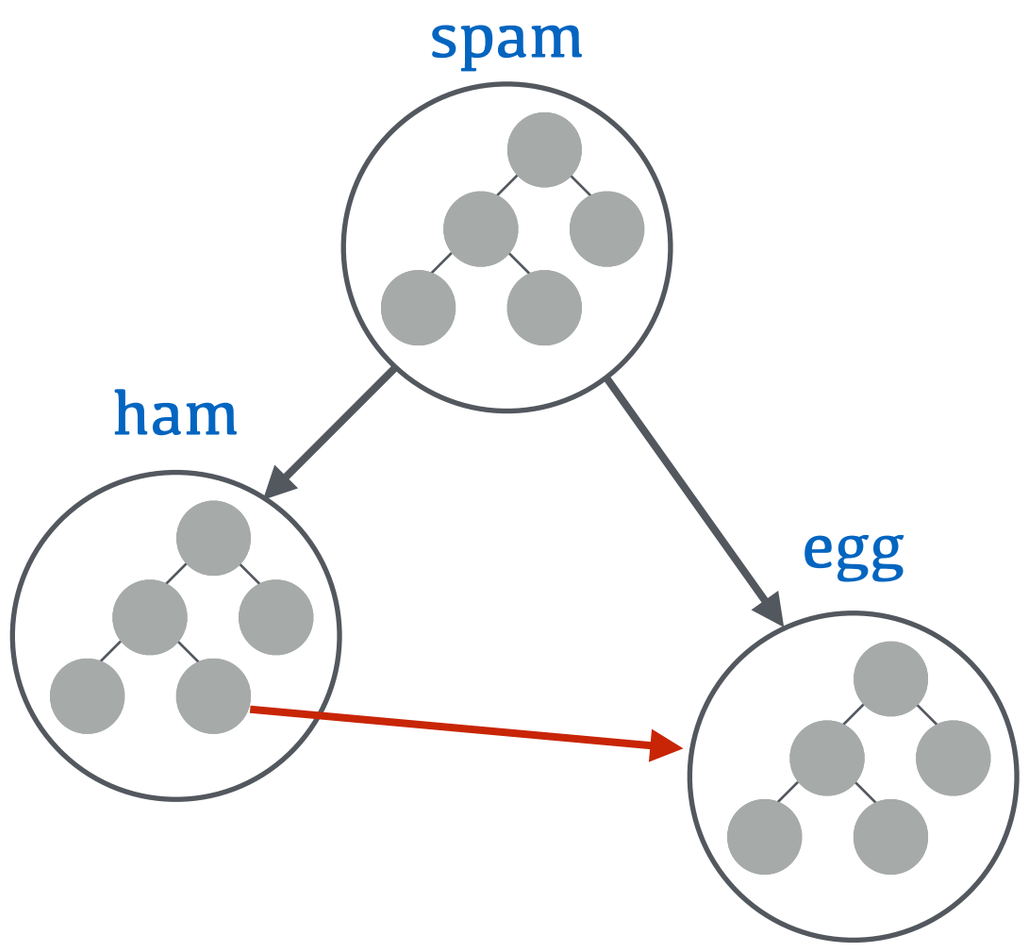



Image from [1]

[1]: T. Würthinger et al., One VM to Rule Them All Onward! 2013

# Call Inlining

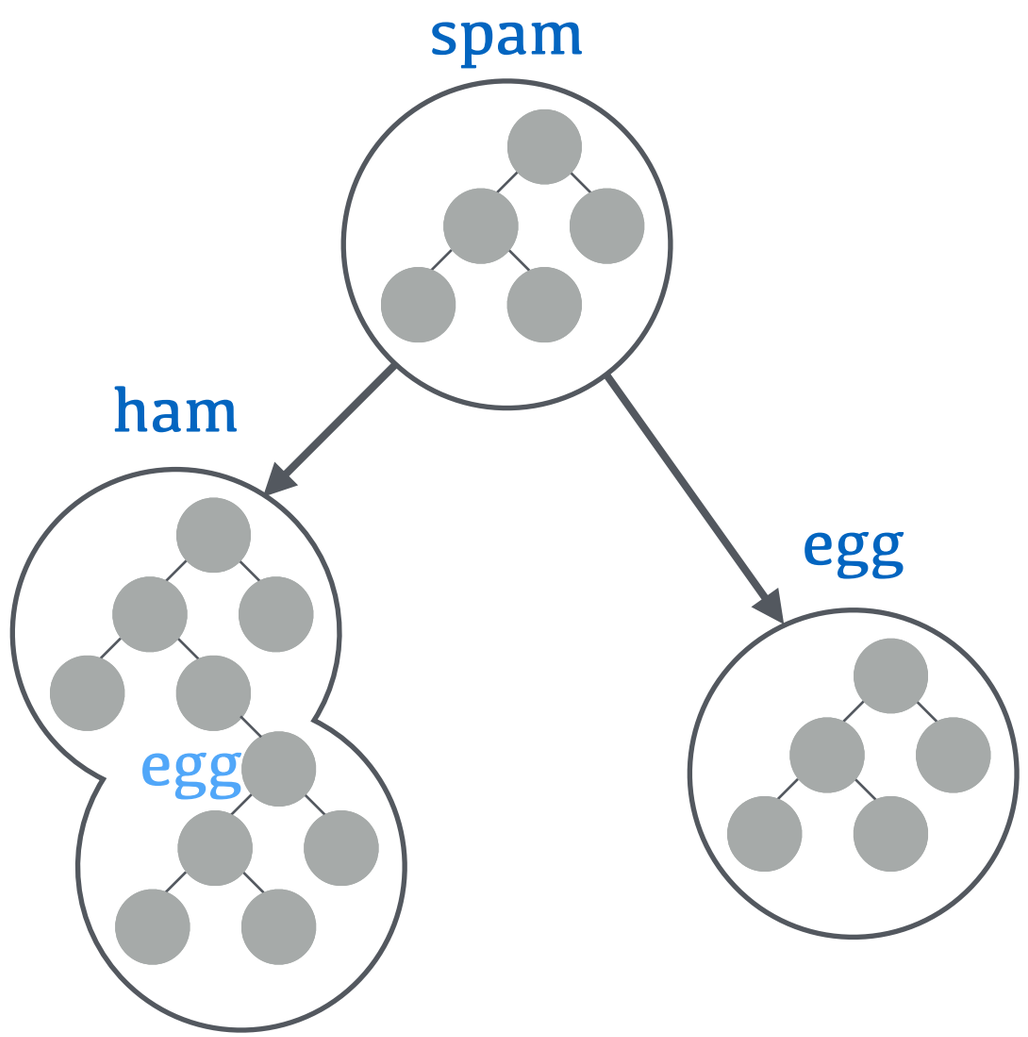



Image from [1]

[1]: T. Würthinger et al., One VM to Rule Them All Onward! 2013

# Example with Call

```
def add(left, right):
  return left + right

def sumitup(n):
  total = 0
  for i in range(n):
    total = add(total, i)
  return total
```

# AST After Parsing

```
FunctionRootNode
  parameters = ParametersOfSizeOneNode
    parameter = WriteLocalUninitializedNode
      rightNode = ReadArgumentNode
  body = BlockNode
    statements[0] = WriteLocalUninitializedNode
      rightNode = IntegerLiteralNode
    statements[1] = ForWithLocalTargetUninitializedNode
      body = BlockNode
        statements[0] = WriteLocalUninitializedNode
          rightNode = UninitializedCallFunctionNode
            callee = ReadGlobalScopeNode
              load = UninitializedLoadAttributeNode
                primary = ObjectLiteralNode
            arguments[0] = ReadLocalUninitializedNode
            arguments[1] = ReadLocalUninitializedNode
      target = WriteLocalUninitializedNode
      iterator = UninitializedCallFunctionNode
        callee = ReadGlobalScopeNode
          load = UninitializedLoadAttributeNode
            primary = ObjectLiteralNode
        arguments[0] = ReadLocalUninitializedNode
    statements[2] = FrameReturnNode
      right = WriteLocalUninitializedNode
        rightNode = ReadLocalUninitializedNode
  returnValue = ReadLocalUninitializedNode
```

# Call Inlined

```
FunctionRootNode
  parameters = ParametersOfSizeOneNode
    parameter = WriteLocalIntNode
      rightNode = ReadArgumentNode
  body = BlockNode
    statements[0] = WriteLocalIntNode
      rightNode = IntegerLiteralNode
    statements[1] = ForWithLocalTargetPRangeNode
      body = BlockNode
        statements[0] = WriteLocalIntNode
          rightNode = CallFunctionNoKeywordInlinedNode
            callee = ReadGlobalDirectNode
              load = LoadObjectAttributeNode
                primary = ObjectLiteralNode
            functionRoot = InlinedFunctionRootNode
              parameters = ParametersOfSizeTwoNode
                param0 = WriteLocalIntNode
                  rightNode = ReadArgumentNode
                param1 = WriteLocalIntNode
                  rightNode = ReadArgumentNode
              body = BlockNode
                statements[0] = FrameReturnNode
                  right = WriteLocalIntNode
                    rightNode = AddIntNode
                      leftNode = ReadLocalIntNode
                      rightNode = ReadLocalIntNode
              returnValue = ReadLocalIntNode
            arguments[0] = ReadLocalIntNode
            arguments[1] = ReadLocalIntNode
      target = WriteLocalIntNode
      iterator = CallBuiltInFunctionDefaultNode
        arguments[0] = ReadLocalIntNode
    statements[2] = FrameReturnNode
      right = WriteLocalIntNode
        rightNode = ReadLocalIntNode
returnValue = ReadLocalIntNode
```

# Machine Code for the Loop

```
            jmp L7

L6:     mov         ecx, edx
        add         ecx, ebp
        jo          L8
        mov         edx, ebp
        incl        edx
        mov         esi, ebp
        mov         ebp, edx
        mov         edx, ecx
L7:     cmp         eax, ebp
        jle         L9
        jmp         L6
L8:     call        deoptimize()

L9:
```

Same as the version without call

# Performance of Our Example

```
def sumitup(n):
  total = 0
  for i in range(n):
    total = total + i
  return total
```

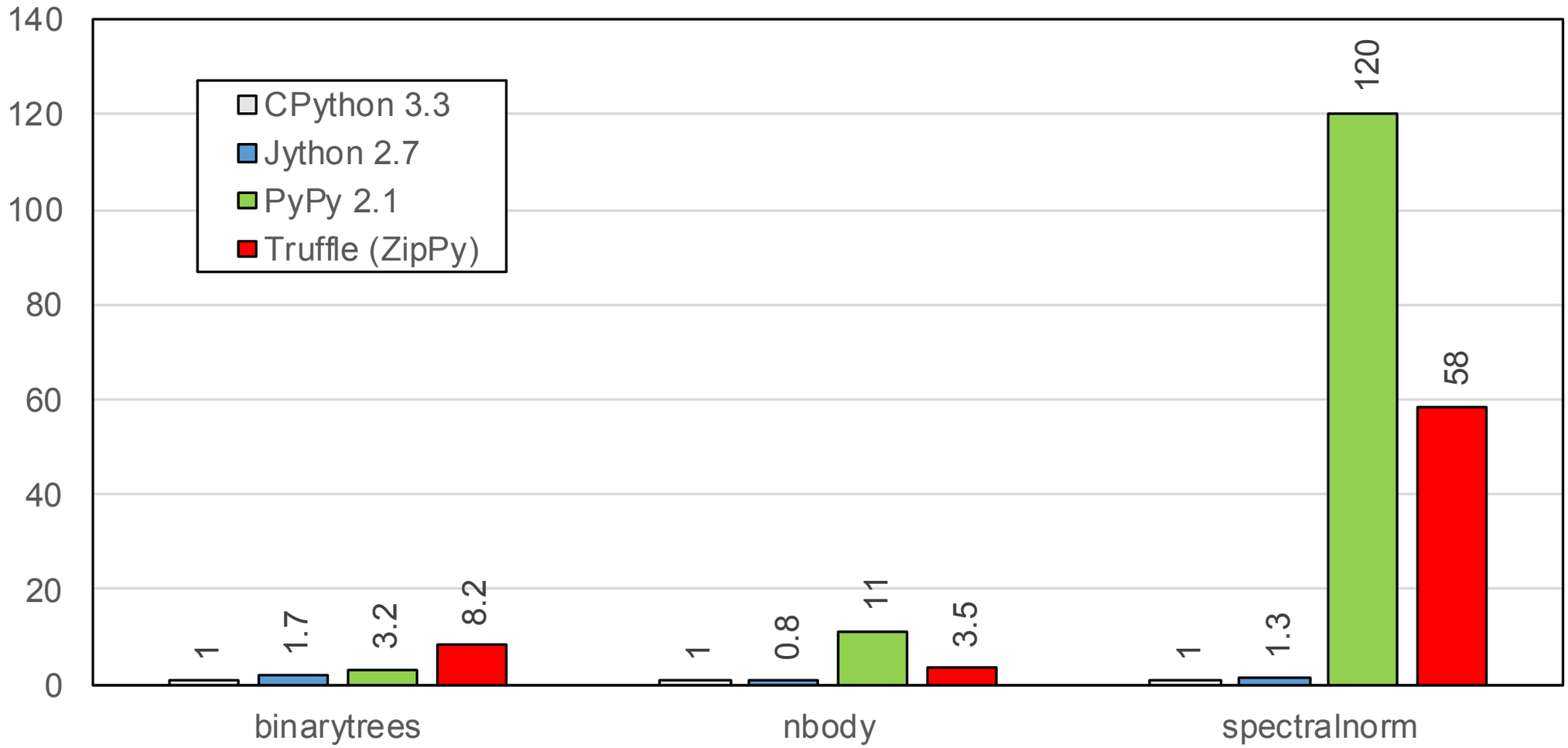
```
def add(left, right):
  return left + right

def sumitup(n):
  total = 0
  for i in range(n):
    total = add(total, i)
  return total
```

50,000 invocations of sumitup(50,000)

| | without call | with call |
|---|---|---|
| CPython 2.7 | 110 sec. | 305 sec. |
| CPython 3.3 | 147 sec. | 330 sec. |
| PyPy 2.1 | 4.0 sec. | 4.4 sec. |
| ZipPy | 3.8 sec. | 3.8 sec. |

Peak performance after warmup runs, so that method is compiled

# Running Benchmarks




Speedup relative to CPython 3.3

# ZipPy

> hg clone https://bitbucket.org/ssllab/zippy

> cd zippy

> ./mx.py build

> ./mx.py python graal/edu.uci.python.benchmark/src/micro/for_range.py

> ./mx.py ideinit

# Thank You

# ZipPy

```
> hg clone https://bitbucket.org/ssllab/zippy

> cd zippy

> ./mx.py build

> ./mx.py python graal/edu.uci.python.benchmark/src/micro/for_range.py

> ./mx.py ideinit
```

**Oracle Labs is looking for summer intern 2014**

**Christian Wimmer**
**<christian.wimmer@oracle.com>**