`invokedynamic` and Jython

Jeremy Siek jeremy.siek@colorado.edu **Shashank Bharadwaj** shashank.bharadwaj@colorado.edu	**Jim Baker** jim.baker@canonical.com
Department of ECEE, University of Colorado, Boulder	Jython Ubuntu Server Team

Outline

- What is Jython?
- Call-site specialization
- Iterator Expression Transformation

Outline

- **What is Jython?**
- Call-site specialization
- Iterator Expression Transformation

What is Jython?

Jython is an implementation of the *Python Programming Language* that compiles to JVM bytecode.

Jython 2.6 will:

- support for Python 2.6
- run only on Java 6+
- improve performance by using invokedynamic

How is Python compiled to Java Bytecode?

Consider the statement:

```
print a + b + c
```

Using `PyObject` as the root-type

The statement:

```
print a + b + c
```

Will be compiled to:

```
PyObject a, b, c;
PyObject tmp = a._add(b)._add(c);
Py.println(tmp);
```

Outline

- What is Jython?
- **Call-site specialization**
- Iterator Expression Transformation

Call-site specialization: Overview

- Replace the current calling convention by using MethodHandles
  - Improves performance
  - Adds flexibility - opens doors for future optimizations

Python `def` compiles to …

```
def f(a, b, c):
    print a + b + c
```

Python `def` compiles to a method

```
def f(a, b, c):
    print a + b + c
```

Is compiled to:

```
public PyObject f$1(PyFrame frame, ThreadState ts){
    PyObject localPyObj = frame.getlocal(0)
                          ._add(frame.getlocal(1))
                          ._add(frame.getlocal(2));
    Py.println(localPyObj);
}
```

- `ThreadState` used to keep track of the system state

Additional bootstrapping required

```
def f(a, b, c):
    print a + b + c
```

At the function definition:

```
PyCode f$1;

f$1 = Py.newCode(paramNames, "f", func_id, paramDefaults, ..);
PyFunction pyFunc = new PyFunction(f$1, ..);
frame.setglobal("f", pyFunc);
```

Different flavors of arg passing in Python

positional/formal arguments	`f(1, 2, 3)`
keyword arguments	`f(b=2, c=3, a=1)`
star/variable arguments	`vargs = (2, 3)` `f(1, *vargs)`
variable keyword arguments	`kwargs = {'a':1, 'b':2, 'c':3}` `f(**kwargs)`

Going to optimize formal args (for now)

positional/formal arguments	`f(1, 2, 3)`
keyword arguments	`f(b=2, c=3, a=1)`
star/variable arguments	`vargs = (2, 3)` `f(1, *vargs)`
variable keyword arguments	`kwargs = {'a':1, 'b':2, 'c':3}` `f(**kwargs)`

Python formal arg call compiles to

```
f(1, 2, 3)
```

Is compiled to:

```
PyInteger _1 = PyInteger.new(1);
PyInteger _2 = PyInteger.new(2);
PyInteger _3 = PyInteger.new(3);

frame.getglobal("f").__call__(_1, _2, _3);
```

Putting it all together

```
public PyObject f$1 (PyFrame f,
                     ThreadState ts) {
    // implementation
}
```

Putting it all together

```
public PyObject f$1 (PyFrame f,
                     ThreadState ts) {
    // implementation
}

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);
```

Putting it all together

```
public PyObject f$1 (PyFrame f,
                     ThreadState ts) {
    // implementation
}

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

frame.getglobal("f")
      .__call__(_1, _2, _3);
```

Putting it all together

```
public PyObject f$1 (PyFrame f,
                     ThreadState ts) {
    // implementation
}

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

frame.getglobal("f")
      .__call__(_1, _2, _3);
```

```
public PyObject call (ThreadState ts,
       PyObject a1, PyObject a2,
       PyObject a3, PyObject closure) {

    PyFrame frame = new PyFrame(this);
    frame.fastlocals[0] = a1;
    frame.fastlocals[1] = a2;
    frame.fastlocals[2] = a3;
    call(ts, frame, closure);
}
```

Putting it all together

```
public PyObject f$1 (PyFrame f,
                     ThreadState ts) {
    // implementation
}

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

frame.getglobal("f")
      .__call__(_1, _2, _3);
```

```
public PyObject call (ThreadState ts,
       PyObject a1, PyObject a2,
       PyObject a3, PyObject closure) {

    PyFrame frame = new PyFrame(this);
    frame.fastlocals[0] = a1;
    frame.fastlocals[1] = a2;
    frame.fastlocals[2] = a3;
    call(ts, frame, closure);
}

public PyObject call(ThreadState ts,
       PyFrame frame, PyObject closure){

    ... // setup frame
    call_function(func_id, frame, ts);
}
```

Current implementation

```
public PyObject f$1 (PyFrame f,
                     ThreadState ts) {
    // implementation
}

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

frame.getglobal("f")
      .__call__(_1, _2, _3);

public PyObject call_function (int func_id,
       PyFrame f, ThreadState ts) {
    switch(func_id) {
        case 1:
            return f$1(f, ts);
        ...
    }
}
```

```
public PyObject call (ThreadState ts,
       PyObject a1, PyObject a2,
       PyObject a3, PyObject closure) {

    PyFrame frame = new PyFrame(this);
    frame.fastlocals[0] = a1;
    frame.fastlocals[1] = a2;
    frame.fastlocals[2] = a3;
    call(ts, frame, closure);
}

public PyObject call(ThreadState ts,
       PyFrame frame, PyObject closure){

    ... // setup frame
    call_function(func_id, frame, ts);
}
```

Implementation has to access frames

```
public PyObject f$1(PyFrame frame, ThreadState ts){
    PyObject localPyObj = frame.getlocal(0)
                          ._add(frame.getlocal(1))
                          ._add(frame.getlocal(2));
    Py.println(localPyObj);
}
```

Using `MethodHandles` for the solution [1]

Create an function which takes in args on stack

```
// optimized version
public PyObject _f$1(PyFrame frame, ThreadState ts, PyObject pyFunc
       PyObject a1, PyObject a2, PyObject a3){
    Py.println(a1._add(a2)._add(a3));
}

public PyObject f$1(PyFrame frame, ThreadState ts){
    _f$1(frame, ts, null, frame.getlocal(0),
                    frame.getlocal(1),
                    frame.getlocal(2));
}
```

Using `MethodHandles` for the solution [2]

At the function `def` point, create and store a methodhandle

```
PyCode f$1;

f$1 = Py.newCode(paramNames, "f", func_id, paramDefaults, ..);

MethodHandle mh = MethodHandles.lookup()
              .findVirtual(getClass(), "_f$1", Py.getMethodType(3));
mh = Py.setupMethodHandle(mh);
mh.bindTo(this);
f$1.mHandle = mh;

PyFunction pyFunc = new PyFunction(f$1, ..);
frame.setglobal("f", pyFunc);
```

Need a dynamic guard, because

```
def foo(arg):
    print 'Hi', arg

while(i > 0):
    foo("JVM")
    def foo(arg, opt="2011")
       print 'Hi', arg, opt
    i -= 1
```

Will produce:

```
Hi JVM
Hi JVM 2011
```

Using `MethodHandles` for the solution [3]

At the call-site call `invokeExact`

```
PyFunction pyFunc = frame.getglobal("f");
pyFunc.func_code.mHandle.invokeExact(ts, pyFunc, _1, _2, _3);
```

Solution: with formal arg passing

```
public PyObject _f$1 (PyFrame f,
  ThreadState ts, PyObject pyFunc,
  PyObject a1, PyObject a2,  PyObject a3) {
    Py.println(a1._add(a2)._add(a3);
}

f$1.mHandle = setupMethodHandle(
       Lookup.findVirtual(getClass()
       "_f$1", Py.getMethodType(3)))
    .bindTo(this);

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

PyFunction pyfunc = frame.getglobal("f");
pyFunc.func_code.mHandle
      .invokeExact(ts, pyFunc, _1, _2, _3);
```

```
public static PyFrame test(ThreadState
ts, PyObject func, PyObject a1,
PyObject a2, PyObject a3){
    // test if this call takes 3 args
}

public static void fallback(ThreadState
ts, PyObject func, PyObject a1,
PyObject a2, PyObject a3){
    func.__call__(ts, a1, a2, a3);
}
```

Solution: with formal arg passing

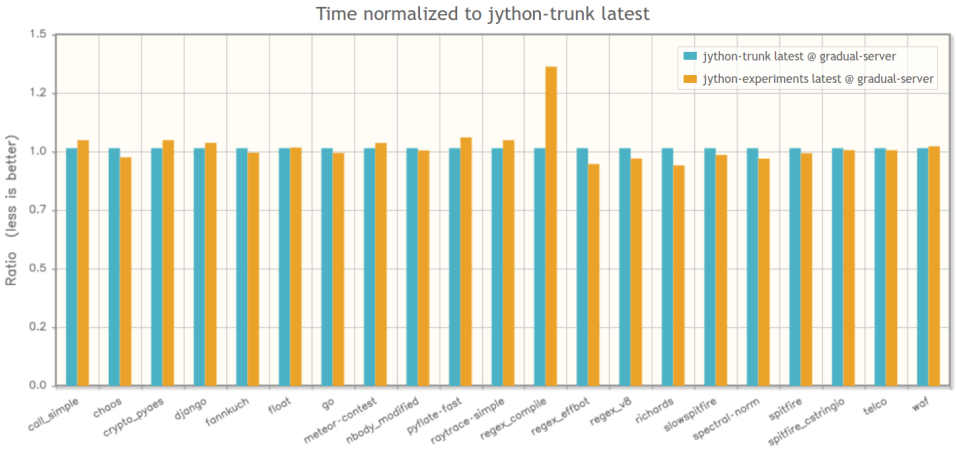
```
public PyObject _f$1 (PyFrame f,
  ThreadState ts, PyObject pyFunc,
  PyObject a1, PyObject a2,  PyObject a3) {
    Py.println(a1._add(a2)._add(a3);
}

f$1.mHandle = setupMethodHandle(
       Lookup.findVirtual(getClass()
       "_f$1", Py.getMethodType(3)))
    .bindTo(this);

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

PyFunction pyfunc = frame.getglobal("f");
pyFunc.func_code.mHandle
      .invokeExact(ts, pyFunc, _1, _2, _3);
```

```
public static PyFrame callBefore
(ThreadState ts, PyObject func){
    // create frame
    PyFrame frame = new PyFrame(..);
    ... // setup frame
    return frame;
}

public static void callAfter(PyObject
ret,
ThreadState ts, PyObject func){
    ... // clean up
}

public static PyObject catchHandler
(Throwable t, ThreadState ts, PyObject
func){
    ... // catch & rethrow PyExeception
}
```

Results: 2.3x improvement in Fibonacci

```
def fibonacci(n):
 if n == 1:
     return 0
 elif n == 2:
     return 1
 else:
     return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(28)
```

Results: Overall benchmark suite



Future

- Extend to kwargs and varargs

Future: Frame creation costly

```
public PyObject _f$1 (PyFrame f,
  ThreadState ts, PyObject pyFunc,
  PyObject a1, PyObject a2,  PyObject a3) {
    Py.println(a1._add(a2)._add(a3);
}

f$1.mHandle = setupMethodHandle(
       Lookup.findVirtual(getClass()
       "_f$1", Py.getMethodType(3)))
    .bindTo(this);

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

PyFunction pyfunc = frame.getglobal("f");
pyFunc.func_code.mHandle
      .invokeExact(ts, pyFunc, _1, _2, _3);
```

```
public static PyFrame callBefore
(ThreadState ts, PyObject func){
    // create frame
    PyFrame frame = new PyFrame(..);
    ... // setup frame
    return frame;
}

public static void callAfter(PyObject
ret,
ThreadState ts, PyObject func){
    ... // clean up
}

public static PyObject catchHandler
(Throwable t, ThreadState ts, PyObject
func){
    ... // catch & rethrow PyExeception
}
```

Future: Frame creation costly, use `@frameless`?

```
public PyObject _f$1 (PyFrame f,
  ThreadState ts, PyObject pyFunc,
  PyObject a1, PyObject a2,  PyObject a3) {
    Py.println(a1._add(a2)._add(a3);
}

f$1.mHandle = setupMethodHandle(
       Lookup.findVirtual(getClass()
       "_f$1", Py.getMethodType(3)))
    .bindTo(this);

PyFunction pyFunc = new PyFunction(f$1, .);
frame.setglobal("f", pyFunc);

PyFunction pyfunc = frame.getglobal("f");
pyFunc.func_code.mHandle
      .invokeExact(ts, pyFunc, _1, _2, _3);
```

```
public static PyFrame callBefore
(ThreadState ts, PyObject func){
    // create frame
    PyFrame frame = new PyFrame(..);
    ... // setup frame
    return frame;
}

public static void callAfter(PyObject
ret,
ThreadState ts, PyObject func){
    ... // clean up
}

public static PyObject catchHandler
(Throwable t, ThreadState ts, PyObject
func){
    ... // catch & rethrow PyExeception
}
```

Outline

- What is Jython?
- Call-site specialization
- **Iterator Expression Transformation**

Iterator Expression Transformation

Consider the `for` loop in Python

```
for i in xrange(1, 10000000):
    sum += (sum ^ i) / i
```

Current implementation is slow

```
PyObject _2 = Py.newInteger(1);
PyObject _3 = Py.newInteger(10000000);
xrangePyObj = frame.getglobal("xrange").__call__(ts, _2, _3).__iter__();
do {
    frame.setlocal(2, itemPyObj);
    PyObject sumPyObj = frame.getlocal(1);
    localPyObj = sumPyObj._iadd(paramPyFrame.getlocal(1)
                                ._xor(paramPyFrame.getlocal(2))
                                ._div(paramPyFrame.getlocal(2)));
    frame.setlocal(1, localPyObj);
    itemPyObj = ((PyObject)xrangePyObj).__iternext__();
} while (itemPyObj != null);
```

Ideally

```
for i in xrange(1000000):
    # do something
```

should correspond to:

```
for (int i=0; i < 1000000; i++) {
    // do something
}
```

- xrange can be dynamically rebound

Using `invokedynamic` to improve performance

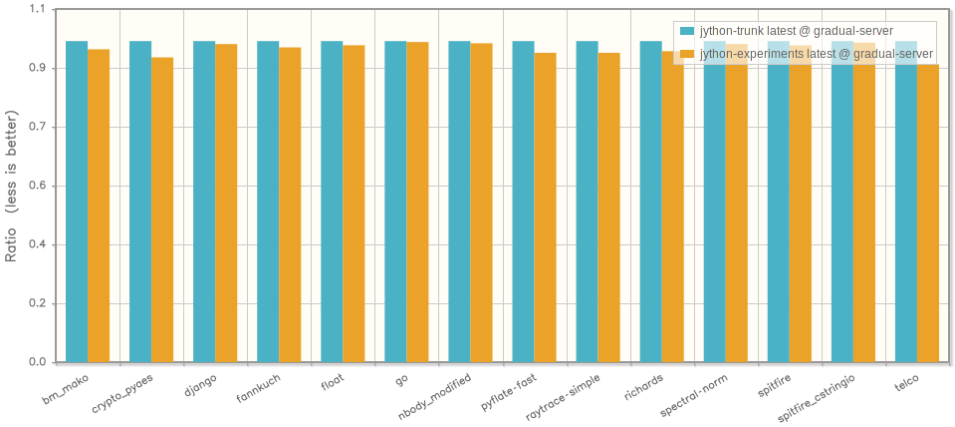
```
try {
     // invokedynamic(frame.getglobal("xrange"));
     int start, stop, step;
     // fill up start, stop and step values

     for(int i = start; i < stop; i+=step){
         iterTmp = Py.newInteger(i);
         // Do the loop stuff here
     }

 } catch (IndyOptimizerException e) {
     // fallback to the previous way of using
     // __iternext__
 }
```

Results

- **10x** improvement on the following micro-benchmark

```
def main():
    sum = 0
    for i in range(1, 10000000):
        sum += (sum ^ i) / i
```

Results: 4% improvement across the suite



Future:

- Handle overflows
- Extend the same logic to:
    - Iterators
    - Decorators

Thank You!

- Project at: http://bitbucket.org/shashank/jython-pilot/
- SpeedCenter at: http://gradual.colorado.edu

Questions?