Python and .Net

Duncan Booth

Overview

- Introduction
- Activestate Study
- Python for .Net
- Approaching the problem
- IronPython
- Issues to be faced
- The Red Queen's Race
- Questions?

Introduction

- 2000
  - Activestate study of Python for .Net
- 2003
  - Brian Lloyd's Python for .Net
  - Interest in native Python on .Net rekindled
- 2004
  - IronPython

Why no Python for .Net?

- Impossible to implement?
  - or just very hard?
- Activestate study deterred people
- Python is a moving target

Why should we want one?

- Getting rid of the Global Interpreter Lock
  - hard to do on C Python
  - opens the door to scalability
- Python fits better in .Net than many of the existing .Net languages
  - .Net is inherently suited to dynamic languages
- It's a platform currently closed to Python

Activestate Study

- Mark Hammond & Greg Stein
- Funded by Microsoft
- 1999 to June 2000
- Often read as saying:
  - A fast implementation of Python on .Net is impossible
- Actually says something more like:
  - This implementation is very slow and we don't know why, but we think it is hard to fix

Conclusions

There is no support for some of the features of .NET that other frameworks will require, such as custom attributes, PInvoke or ASP.NET.

Related topics are the mismatch between the class/instance semantics, module/package semantics and the exception systems.

There is no support for some Python features that some programs will require. Examples include: string formatting, long integers, complex numbers, standard library, etc.

Conclusions

- *The speed of the current system is so low as to render the current implementation useless for anything beyond demonstration purposes.*
- *...*
- *Some of the blame for this slow performance lies in the domain of .NET internals and Reflection::Emit, but some of it is due to the simple implementation of the Python for .NET compiler.*

Conclusions

- *Only a small amount of effort has gone into analysing the performance of the runtime, mainly due to the lack of performance analysis tools available for .NET. Without such tools, making performance related changes is fruitless, as the effectiveness is difficult to measure.*

Conclusions

- *Not withstanding the tuning of the runtime system, the simple existence of the runtime accounts for much of our performance problem. When simple arithmetic expressions take hundreds or thousands of Intermediate Language instructions (via the Python runtime) to complete, performance will always be a struggle.*

Polymorphism for arithmetic

- a = 42
- b = a + 1
  - compiles to:

    a.__radd__(1)
- c = a + b
  - compiles to:

    a.__add__(b)
  - which calls b.__radd__(42)
- In **principle**, simple arithmetic could be
  - one or two virtual method calls +
  - the operation itself +
  - the creation of the result.

Polymorphism for arithmetic

```
class PyObject {
    public virtual PyObject __add__(PyObject p) { ... }
    public virtual PyObject __radd__(PyObject p) { ... }

    public virtual PyObject __radd__(int v) {
        return this.__radd__((PyInt)v);
    }
};
class PyInt: PyObject {
    override PyObject __add__(PyObject p) {
        return p.__radd__(this.value);
    }
    override PyObject __radd__(int v) {
        try {
            return (PyInt) checked(this.value + v);
        } catch(OverflowException) {
            return (PyLong)((long)this.value + v);
        }
    }
}
```

Activestate Study Limitations

- Modelled C Python API
  - not object oriented
  - functions had to test the types of their arguments
- PyObject was a value type
  - so no use of inheritance/polymorphism

Python for .Net

- Implementation by Brian Lloyd
- Bridges C-Python to the CLR
- You can:
  - run existing Python scripts
  - use existing libraries
- and:
  - import CLR classes
  - access attributes and methods
  - subclass a CLR class?
  - partly works under Mono

Python for .Net

- You cannot:
  - use a Python class from the CLR
  - script an ASP.Net page

Value types vs Reference types

```
items = CLR.System.Array.CreateInstance(Point, 3)
for i in range(3):
    items[i] = Point(0, 0)

items[0].X = 1 # won't work!!
```

Value types vs Reference types

```
items = CLR.System.Array.CreateInstance(Point, 3)
for i in range(3):
    items[i] = Point(0, 0)

# This _will_ work.

item = items[0]
item.X = 1
items[0] = item
```

- This is probably going to be an issue with any Python running under the CLR

How fast is it?

- Simple assignment into a hashtable
  - slower by a factor of 100
- Every call across the bridge is a bottleneck
  - this is very similar to Python and COM
- But:
  - Python part of code runs at normal C Python speed
  - CLR code runs at normal CLR speed

Approaching the problem

- July 2003 onwards
  - I re-read the Activestate paper
  - Got the prototype running on .Net 1.0
  - Pystone about 20-25 times slower than Python 2.3
  - Refactored to use reference classes for Python objects
  - Various micro optimisations
  - Finally realised where the bottleneck lies
    - (every function call used reflection)
  - Plans shelved when ...

IronPython

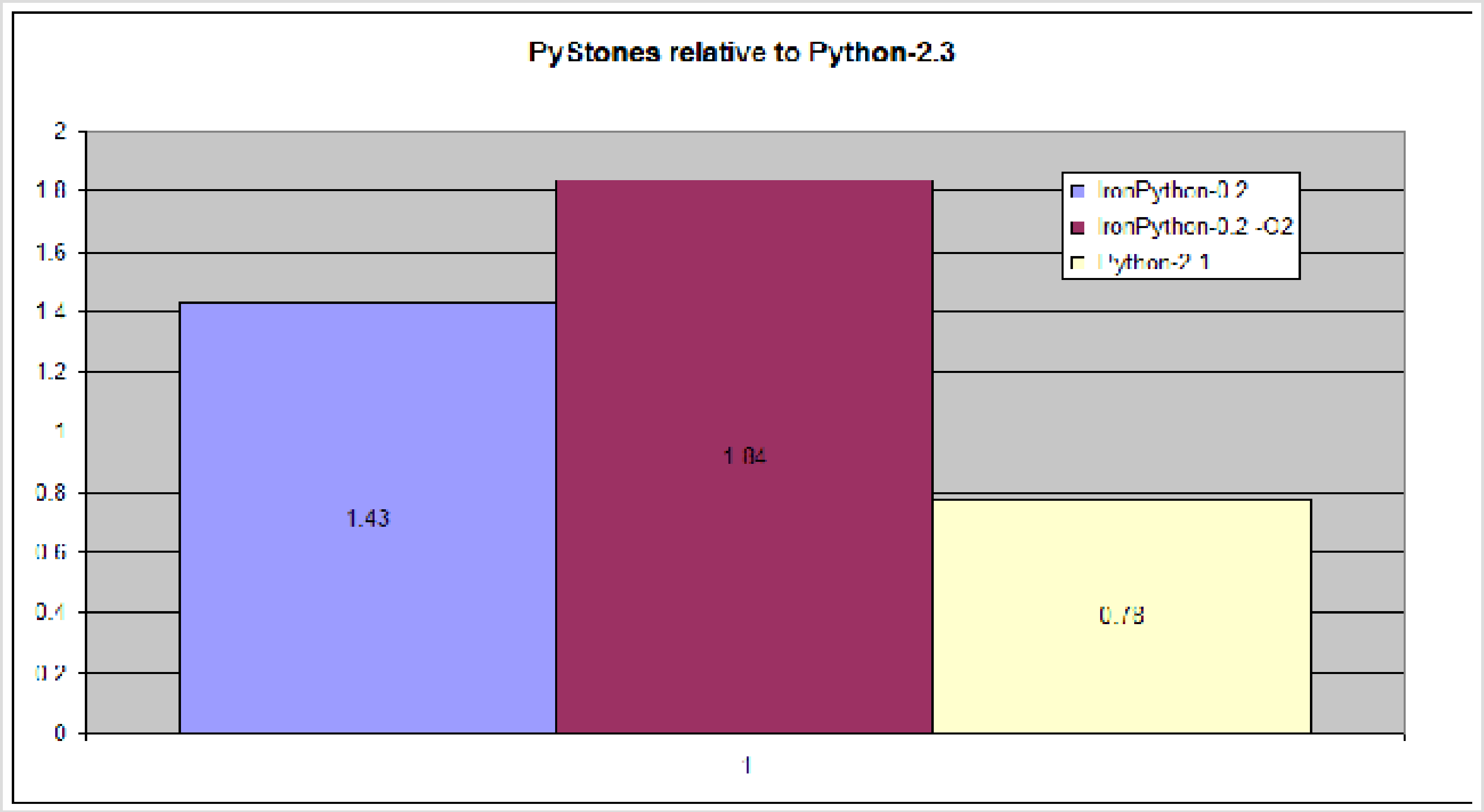
- Being Developed by Jim Hugunin
- Fast
  - IronPython-0.2 is 1.4x faster than Python-2.3 on the standard pystone benchmark.
- Integrated with the Common Language Runtime
  - IronPython code can easily use CLR libraries and Python classes can extend CLR classes.
- Fully dynamic
  - IronPython supports an interactive interpreter and transparent on-the-fly compilation of source files just like standard Python.

IronPython

- Optionally static
  - IronPython also supports static compilation of Python code to produce static executables (.exe's) or static libraries (.dll's)
- Managed and verifiable
  - IronPython generates verifiable assemblies with no dependencies on native libraries that can run in environments which require verifiable managed code.
- Under development
  - IronPython is currently an unreleased research prototype.

(bullet points from http://ironpython.com)

IronPython



from http://www.python.org/pycon/dc2004/papers/9

Issues to be faced

- N.B.
  - The following slides are based on my experiences
  - The solutions used in IronPython may be very different

The problem with delegates

- The CLR does not permit use of function pointers in validated code
- Delegates can replace function pointers for:
  - bound methods
  - static functions
- BUT
  - there is no delegate equivalent to unbound methods
  - also delegates cannot point to constructors

Why unbound methods matter

- Using Reflection to find and call a method on an object is **slow**.
  - by about a factor of 100 (again)

- Alternative to Reflection is:
  - create unbound method objects and store them in a dictionary on the class.
  - find and call method by:
    - dictionary lookup (or simple string comparison)
    - then either call directly or create bound method.

Solution: Generate wrappers

- Given a class:

```
class Foo {
   public int bar(string s) { ... }
}
```

- compile a wrapper automatically:

```
class FooClass {
   static PyObject bar(PyObject self, PyObject s) {
       return (PyInt)(
             ((Foo)self).bar((string)s));
   };
}
```

- __getattribute__ defined to create an unbound method object with a delegate to the static bar.

Calling a function is slow?

- C-Python function calls use generic prototype:
  - Functions receive:
    - tuple of positional arguments
    - dictionary of keyword arguments
  - Arguments have to be unscrambled in most general way
- BUT:
  - Python functions always expect a fixed number of arguments
  - Calls usually pass a fixed number of arguments

Calling functions

- These functions all expect 3 arguments
  - (although they may be called in quite different ways):

```
def f1(a=0, b=1, c=2): pass
def f2(a, *b, **c): pass
def f3(a, (p, q), c): pass
```

- Compiling a call, we know
  - how many arguments are passed
  - not: how many are expected
- Use polymorphism to optimise the call

Calling functions

```
def f1(a=0, b=1, c=2): ...
```

- The def statement compiles to (roughly):

```
f1 = new PyFunction3(new PyFunction3Delegate(f1_code),
    new string[] { “a”, “b”, “c” },
    new PyObject[] { (PyInt)0, (PyInt)1, (PyInt)2 })
```

- A call: f1(x) compiles to:

```
f1.__call__(x)
```

  - Implementation:

```
return this.delegate(arg1, this.defaults[1],
  this.defaults[2]);
```

Calling Functions

- __call__ is overloaded:
  - for 0..n arguments
  - for more than n arguments
  - for keyword arguments
- Also need n+1 classes for:
  - PyFunction0..n (and many or keywords)
  - PyBoundMethod0..n-1 (and many or keywords)
  - etc.
- n had better not be too large!

Optimising global variables

- Setting a global should simply set a static variable
- Accessing a global should retrieve the static variable
- globals() returns a dictionary-like object
  - __getitem__, __setitem__ for known variables refer to static values
  - unknown global variables are held in the dictionary
- Updates to module variables from outside module done through globals() pseudo-dictionary

Optimising builtins

- References to builtins may be optimised at compile time
  - but **only** if no global is set to explicitly hide the builtin
- Indirectly setting a global that masks an optimised builtin should raise an exception
- Globals set once (e.g. by import) could also be used for optimisations (and made readonly)

Optimising Globals/Builtins

```
from CLR.Python import overloads
from CLR.System import Int32, String

class C:
   def foo(a, b=””) [
       overloads((Int32,), (Int32, String)) ]:
       pass
```

(subject to PEP318 ever settling down, the exact code could look different)

- This might
  - generate a class usable by other CLR languages
  - with two overloaded public 'foo' methods.
- and
  - reject any attempt to redefine 'overloads' or 'C' in the module

Optimising Globals/Builtins

```
from CLR.Python import overloads
from CLR.System import Int32, String

class C:
   [overloads((Int32,), (Int32, String))]
   def foo(a, b=””) :
       pass
```

(subject to PEP318 ever settling down, the exact code could look different)

- This might
  - generate a class usable by other CLR languages
  - with two overloaded public 'foo' methods.
- and
  - reject any attempt to redefine 'overloads' or 'C' in the module

The Red Queen's Race

- IronPython needs:
  - first public release (in any form)
  - development of runtime and libraries
  - bringing in line with Python 2.4
    - generator comprehensions
    - decorators
    - relative imports
- Can it get there?
  - Maybe only if Python slows down

The Red Queen's Race

"A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

"I'd rather not try, please!" said Alice. "I'm quite content to stay here--only I am so hot and thirsty!"

Questions?