Faster than C#: efficient implementation of dynamic languages on .NET

Antonio Cuni
Davide Ancona
Armin Rigo

ICOOOLPS@ECOOP 2009 - Genova, Italy

July 6, 2009

Introduction

- Dynamic languages are nice
  - e.g., Python
- so are .NET and the JVM
- Problem: slow!
- Solution: make them faster :-)
- We concentrate our efforts on .NET

State of the art

- IronPython
- Jython
- JRuby, Groovy, ...

- **Self**
- Javascript: TraceMonkey, V8
- ...

State of the art

- IronPython
- Jython
- JRuby, Groovy, ...
- **Self**
- Javascript: TraceMonkey, V8
- ...

Why so slow?

- Hard to compile efficiently
- Lack of type information at compile-time
- VMs not optimized to run them
- .NET is a multi-language VM?
  - Sure, as long as the language is C#
- JVM is in a better shape, but still heavily optimized for Java

Why so slow?

- Hard to compile efficiently
- Lack of type information at compile-time
- VMs not optimized to run them
- .NET is a multi-language VM?
  - Sure, as long as the language is C#
- JVM is in a better shape, but still heavily optimized for Java

Why so slow?

- Hard to compile efficiently
- Lack of type information at compile-time
- VMs not optimized to run them
- .NET is a multi-language VM?
  - Sure, as long as the language is C#
- JVM is in a better shape, but still heavily optimized for Java

JIT compiler

- Wait until you know what you need
- Interweave compile-time and runtime
- Exploit runtime information

JIT on top of .NET

- JIT layering
- How to extend existing code?
- Fight the VM

JIT compiler

- Wait until you know what you need
- Interweave compile-time and runtime
- Exploit runtime information

JIT on top of .NET

- JIT layering
- How to extend existing code?
- Fight the VM

PyPy

- Python in Python
- (lots of features and goals)
- **JIT compiler generator**
- Python semantics for free
- JIT frontend
  - Not limited to Python
- JIT backends
  - x86 backend
  - **CLI/.NET backend**

- Note: this talk is about JIT v2

PyPy

- Python in Python
- (lots of features and goals)
- **JIT compiler generator**
- Python semantics for free
- JIT frontend
  - Not limited to Python
- JIT backends
  - x86 backend
  - **CLI/.NET backend**

- Note: this talk is about JIT v2

Partial evaluation (PE)

- Assume the Python bytecode to be constant
- Constant-propagate it into the Python interpreter.
- Colors
  - Green: compile-time value
  - Red: runtime value

Partial Evaluation with Colors

- Green operations: unchanged, executed at compile-time
- Red operations: converted into corresponding code emitting code

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

Partial Evaluation with Colors

- Green operations: unchanged, executed at compile-time
- Red operations: converted into corresponding code emitting code

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

Partial Evaluation with Colors

- Green operations: unchanged, executed at compile-time
- Red operations: converted into corresponding code emitting code

Example

```
def f(x, y):
  x2 = x * x
  y2 = y * y
  return x2 + y2
```

case x=10

```
def f_10(y):
  y2 = y * y
  return 100 + y2
```

Challenges

- A shortcoming of PE is that in many cases not much can be really assumed constant at compile-time: poor results
- Effective dynamic compilation requires feedback of runtime information into compile-time
- For a dynamic language: types are a primary example

Solution: Promotion

- “Promote” run-time values to compile-time
- Promotion guided by few hints in the interpreter
- Stop the compilation at promotions
- Execute until promotion points
- Compile more

Promotion (example)

Example

```
def f(x, y):
  x1 = hint(x, promote=True)
  return x1*x1 + y*y
```

original

```
def f_(x, y):
  switch x:
    pass
  default:
    compile_more(x)
```

augmented

```
def f_(x, y):
  switch x:
    case 3:
      return 9 + y*y
  default:
    compile_more(x)
```

Promotion (example)

Example

```
def f(x, y):
  x1 = hint(x, promote=True)
  return x1*x1 + y*y
```

original

```
def f_(x, y):
  switch x:
    pass
  default:
    compile_more(x)
```

augmented

```
def f_(x, y):
  switch x:
    case 3:
      return 9 + y*y
  default:
    compile_more(x)
```

Promotion (example)

Example

```
def f(x, y):
  x1 = hint(x, promote=True)
  return x1*x1 + y*y
```

original

```
def f_(x, y):
  switch x:
    pass
  default:
    compile_more(x)
```

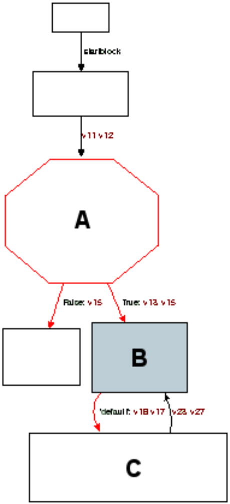
augmented

```
def f_(x, y):
  switch x:
    case 3:
      return 9 + y*y
  default:
    compile_more(x)
```

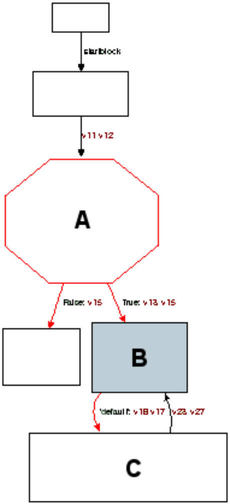
Promotion on .NET

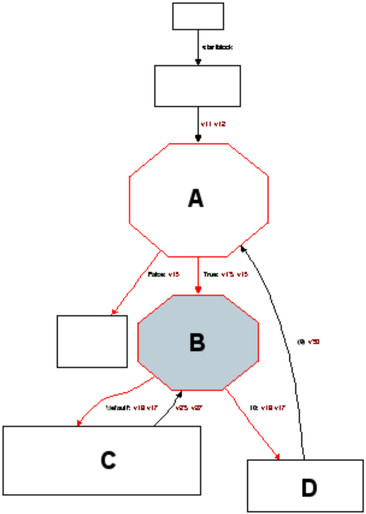
- Flexswitch
  - Growable switch
  - Can add new cases at runtime
- Ideally as efficient as a jump
- No support from the VM
- Very costly
- Still effective as long as it's not in the hot path

Flexswitch example



Flexswitch example





Flexswitch for CLI

- Unit of compilation: method
- Flowgraphs split into multiple methods
- Primary method
  - Contains a trampoline
  - Array of delegates
- Secondary methods
  - Stored into that array
- Jumps between secondary methods go through the trampoline
- Hard (and slow!) to pass arguments around

TLC

- Python not (yet) supported :-(
- Dynamic toy language
- Designed to be “as slow as Python”
- Stack manipulation
- Boxed integers
- Dynamic lookup of methods

Benchmarks (1)

Factorial

n	10	10^7	10^8	10^9
Interp	0.031	30.984	N/A	N/A
JIT	0.422	0.453	0.859	4.844
JIT 2	0.000	0.047	0.453	4.641
C#	0.000	0.031	0.359	3.438
Interp/JIT 2	N/A	**661.000**	N/A	N/A
JIT 2/C#	N/A	**1.500**	**1.261**	**1.350**

Benchmarks (2)

Fibonacci

n	10	10^7	10^8	10^9
Interp	0.031	29.359	N/A	N/A
JIT	0.453	0.469	0.688	2.953
JIT 2	0.000	0.016	0.250	2.500
C#	0.000	0.016	0.234	2.453
Interp/JIT 2	N/A	**1879.962**	N/A	N/A
JIT 2/C#	N/A	**0.999**	**1.067**	**1.019**

Benchmars (3)

```
def main(n):
    if n < 0:
        n = -n
        obj = new(value, accumulate=count)
    else:
        obj = new(value, accumulate=add)
    obj.value = 0
    while n > 0:
        n = n - 1
        obj.accumulate(n)
    return obj.value

def count(x):
    this.value = this.value + 1
def add(x):
    this.value = this.value + x
```

Benchmars (4)

Accumulator

n	10	10^7	10^8	10^9
Interp	0.031	43.063	N/A	N/A
JIT	0.453	0.516	0.875	4.188
JIT 2	0.000	0.047	0.453	3.672
C#	0.000	0.063	0.563	5.953
Interp/JIT 2	N/A	**918.765**	N/A	N/A
JIT 2/C#	N/A	**0.750**	**0.806**	**0.617**

Future work

- Non local jumps are terribly slow
- Good results only if they are not in the inner loop
- Recompile hot non-local jumps?
- Tracing JIT?
  - You have just seen it in the previous talk :-)

Contributions

- JIT layering works
  - Optimize different levels of overhead
  - .NET's own JIT could be improved
- Current VMs are limited
  - How to make them more friendly to dynamic languges?