PyPy's Approach to Implementing Dynamic Languages Using a Tracing JIT Compiler

Carl Friedrich Bolz

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

IBM Watson Research Center, February 8th, 2011

Scope

This talk is about:

- implementing dynamic languages
  (with a focus on complicated ones)
- in a context of limited resources
  (academic, open source, or domain-specific)
- imperative, object-oriented languages
- single-threaded implementations

Scope

This talk is about:

- implementing dynamic languages
  (with a focus on complicated ones)
- in a context of limited resources
  (academic, open source, or domain-specific)
- imperative, object-oriented languages
- single-threaded implementations

Goals

Reconciling:

- flexibility, maintainability (because languages evolve)
- simplicity (because teams are small)
- performance

Outline

1. The Difficulties of Implementing Dynamic Languages
    - Technical Factors
    - Requirements

Outline

1. The Difficulties of Implementing Dynamic Languages
   - Technical Factors
   - Requirements
2. Approaches For Dynamic Language Implementation
   - Implementing VMs in C/C++
   - Method-Based JIT Compilers
   - Tracing JIT Compilers
   - Building on Top of an OO VM

Outline

What is Needed Anyway

A lot of things are not really different from other languages:

- lexer, parser
- (bytecode) compiler
- garbage collector
- object system

Control Flow

- every language implementation needs a way to implement the control flow of the language
- trivially and slowly done in interpreters with AST or bytecode
- technically very well understood
- sometimes small difficulties, like generators in Python

Control Flow

- every language implementation needs a way to implement the control flow of the language
- trivially and slowly done in interpreters with AST or bytecode
- technically very well understood
- sometimes small difficulties, like generators in Python
- some languages have more complex demands but this is rare
- examples: Prolog

Late Binding

- lookups can be done only at runtime
- historically, dynamic languages have moved to ever later binding times
- a large variety of mechanisms exist in various languages
- mechanism are often very ad-hoc because "it was easy to do in an interpreter"

Late Binding in Python

In Python, the following things are late bound

- global names
- modules
- instance variables
- methods

Late Binding in Python

In Python, the following things are late bound

- global names
- modules
- instance variables
- methods
- the class of objects
- class hierarchy

Dispatching

- dispatching is a very important special case of late binding
- how are the operations on objects implemented?
- this is usually very complex, and different between languages

Dispatching

- dispatching is a very important special case of late binding
- how are the operations on objects implemented?
- this is usually very complex, and different between languages
- operations are internally split up into one or several lookup and call steps
- a huge space of paths ensues
- most of the paths are uncommon

Example: Attribute Reads in Python

What happens when an attribute `x.m` is read? (simplified)

- check for the presence of `x.__getattribute__`, if there, call it

Example: Attribute Reads in Python

What happens when an attribute `x.m` is read? (simplified)

- check for the presence of `x.__getattribute__`, if there, call it
- look for the name of the attribute in the object's dictionary, if it's there, return it

Example: Attribute Reads in Python

What happens when an attribute `x.m` is read? (simplified)

- check for the presence of `x.__getattribute__`, if there, call it
- look for the name of the attribute in the object's dictionary, if it's there, return it
- walk up the MRO of the object and look in each class' dictionary for the attribute

Example: Attribute Reads in Python

What happens when an attribute `x.m` is read? (simplified)

- check for the presence of `x.__getattribute__`, if there, call it
- look for the name of the attribute in the object's dictionary, if it's there, return it
- walk up the MRO of the object and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result

Example: Attribute Reads in Python

What happens when an attribute `x.m` is read? (simplified)

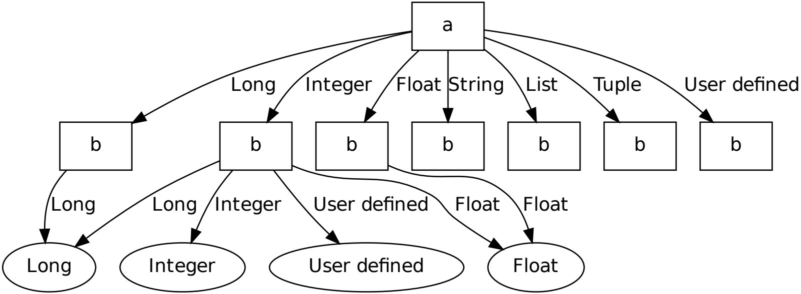
- check for the presence of `x.__getattribute__`, if there, call it
- look for the name of the attribute in the object's dictionary, if it's there, return it
- walk up the MRO of the object and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result
- if the attribute is not found, look for `x.__getattr__`, if there, call it

Example: Attribute Reads in Python

What happens when an attribute `x.m` is read? (simplified)

- check for the presence of `x.__getattribute__`, if there, call it
- look for the name of the attribute in the object's dictionary, if it's there, return it
- walk up the MRO of the object and look in each class' dictionary for the attribute
- if the attribute is found, call its `__get__` attribute and return the result
- if the attribute is not found, look for `x.__getattr__`, if there, call it
- raise an `AttributeError`

Example: Addition in Python



Dependencies Between Subsequent Dispatches

- one dispatch operation is complex
- many in a sequence are worse
- take `(a + b) + c`
- the dispatch decision of the first operation influences the second

Boxing of Primitive Values

- primitive values often need to be boxed,
  to ensure uniform access
- a lot of pressure is put on the GC by arithmetic
- need a good GC (clear anyway)
- in arithmetic, lifetime of boxes is known

Escaping Paths

- considering again `(a + b) + c`
- assume `a` and `b` are ints
- then the result should not be allocated
- escaping path: if `c` has a user-defined class

(Frames)

- side problem:
- many languages have reified frame access
- e.g. Python, Smalltalk, Ruby, ...
- support for in-language debuggers
- in an interpreter these are trivial, because the interpreter needs them anyway
- how should reified frames work efficiently when a compiler is used?

Summarizing the Requirements

1. control flow
2. late binding
3. **dispatching**
4. **dependencies between subsequent dispatches**
5. boxing
6. (reified frames)

Common Approaches to Language Implementation

- Using C/C++
  - for an interpreter
  - for a static compiler
  - for a method-based JIT
  - for a tracing JIT
- Building on top of a general-purpose OO VM

Common Approaches to Language Implementation

Using C/C++

- CPython (interpreter)
- Ruby (interpreter)
- V8 (method-based JIT)
- TraceMonkey (tracing JIT)
- ...

Common Approaches to Language Implementation

Using C/C++

- CPython (interpreter)
- Ruby (interpreter)
- V8 (method-based JIT)
- TraceMonkey (tracing JIT)
- ...

Building on top of a general-purpose OO VM

- Jython, IronPython
- JRuby, IronRuby
- various Prolog, Lisp, even Smalltalk implementations

Implementing VMs in C

When writing a VM in C it is hard to reconcile our goals

- flexibility, maintainability
- simplicity
- performance

Implementing VMs in C

When writing a VM in C it is hard to reconcile our goals

- flexibility, maintainability
- simplicity
- performance

Python Case

- CPython is a very simple bytecode VM, performance not great
- Psyco is a just-in-time-specializer, very complex, hard to maintain, but good performance
- Stackless is a fork of CPython adding microthreads. It was never incorporated into CPython for complexity reasons

Interpreters in C/C++

- mostly very easy
- well understood problem
- portable, maintainable
- slow

How do Interpreters Meet the Requirements?

	Interpreter	Static Compiler	Method Compiler	Tracing JIT	OO VMs
Control Flow	-				
Late Binding	-				
Dispatching	-				
Dependencies	-				
Boxing	-				
(Reified Frames)	-				

Static Compilers to C/C++

- first reflex of many people is to blame it all on bytecode dispatch overhead
- thus static compilers are implemented that reuse the object model of an interpreter
- gets rid of interpretation overhead only
- seems to give about 2x speedup

Static Compilers to C/C++

- first reflex of many people is to blame it all on bytecode dispatch overhead
- thus static compilers are implemented that reuse the object model of an interpreter
- gets rid of interpretation overhead only
- seems to give about 2x speedup
- dispatch, late-binding and boxing only marginally improved
- static analysis mostly never works

Static Compilers to C/C++

- first reflex of many people is to blame it all on bytecode dispatch overhead
- thus static compilers are implemented that reuse the object model of an interpreter
- gets rid of interpretation overhead only
- seems to give about 2x speedup
- dispatch, late-binding and boxing only marginally improved
- static analysis mostly never works

Python Case

- Cython, Pyrex are compilers from large subsets of Python to C
- lots of older experiments, most discontinued

How do Static Compilers Meet the Requirements?

	Interpreter	**Static Compiler**	Method Compiler	Tracing JIT	OO VMs
Control Flow	-	+			
Late Binding	-	-			
Dispatching	-	-			
Dependencies	-	-			
Boxing	-	-			
(Reified Frames)	-	--			

Method-Based JIT Compilers

- to fundamentally attack some of the problems, a dynamic compiler is needed
- a whole new can of worms

Method-Based JIT Compilers

- to fundamentally attack some of the problems, a dynamic compiler is needed
- a whole new can of worms
  - type profiling
  - inlining based on that
  - general optimizations
  - complex backends
- very hard to pull off for a volunteer team

Method-Based JIT Compilers

- to fundamentally attack some of the problems, a dynamic compiler is needed
- a whole new can of worms
    - type profiling
    - inlining based on that
    - general optimizations
    - complex backends
- very hard to pull off for a volunteer team

Examples

- Smalltalk and SELF JITs
- V8 and JägerMonkey
- Psyco, sort of

Compilers are a bad encoding of Semantics

- to improve all complex corner cases of the language, a huge effort is needed
- often needs a big "bag of tricks"
- the interactions between all tricks is hard to foresee
- the encoding of language semantics in the compiler is thus often obscure and hard to change

Compilers are a bad encoding of Semantics

- to improve all complex corner cases of the language, a huge effort is needed
- often needs a big "bag of tricks"
- the interactions between all tricks is hard to foresee
- the encoding of language semantics in the compiler is thus often obscure and hard to change

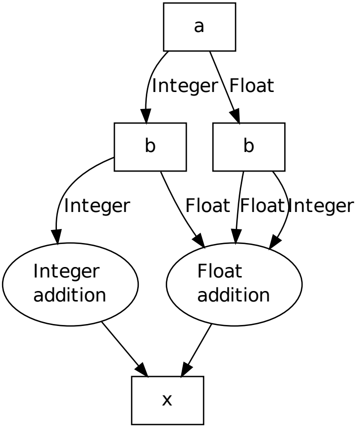
Python Case

- Psyco is a dynamic compiler for Python
- synchronizing with CPython's development is a lot of effort
- many of CPython's new features not supported well
- not ported to 64-bit machines, and probably never will

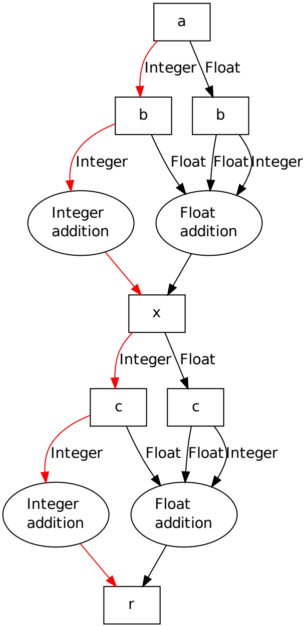
Method-Based JITs and Dispatching Dependencies

```
x = add(a, b)
r = add(x, c)
```

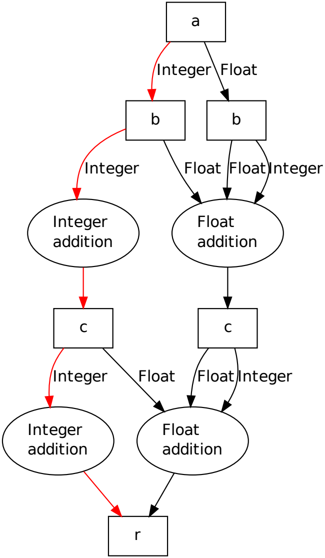
Method-Based JITs and Dispatching Dependencies



Method-Based JITs and Dispatching Dependencies



Method-Based JITs and Dispatching Dependencies



How do Method-Based JIT Compilers Meet the Requirements?

	Interpreter	Static Compiler	**Method Compiler**	Tracing JIT	OO VMs
Control Flow	-	+	+		
Late Binding	-	-	+		
Dispatching	-	-	+		
Dependencies	-	-	?		
Boxing	-	-	?		
(Reified Frames)	-	--	?		

Tracing JIT Compilers

- relatively recent approach to JIT compilers
- pioneered by Michael Franz and Andreas Gal for Java
- turned out to be well-suited for dynamic languages

Tracing JIT Compilers

- relatively recent approach to JIT compilers
- pioneered by Michael Franz and Andreas Gal for Java
- turned out to be well-suited for dynamic languages

Examples

- TraceMonkey
- LuaJIT
- SPUR, sort of
- PyPy, sort of

Tracing JIT Compilers

- idea from Dynamo project:
  dynamic rewriting of machine code
- conceptually simpler than type profiling

Tracing JIT Compilers

- idea from Dynamo project:
  dynamic rewriting of machine code
- conceptually simpler than type profiling

Basic Assumption of a Tracing JIT

- programs spend most of their time executing loops
- several iterations of a loop are likely to take similar code paths

Tracing VMs

- mixed-mode execution environment
- at first, everything is interpreted
- lightweight profiling to discover hot loops
- code generation only for common paths of hot loops
- when a hot loop is discovered, start to produce a trace

Tracing

- a trace is a sequential list of operations
- a trace is produced by recording every operation the interpreter executes
- tracing ends when the tracer sees a position in the program it has seen before
- a trace thus corresponds to exactly one loop
- that means it ends with a jump to its beginning

Tracing

- a trace is a sequential list of operations
- a trace is produced by recording every operation the interpreter executes
- tracing ends when the tracer sees a position in the program it has seen before
- a trace thus corresponds to exactly one loop
- that means it ends with a jump to its beginning

Guards

- the trace is only one of the possible code paths through the loop
- at places where the path could diverge, a guard is placed

Code Generation and Execution

- being linear, the trace can easily be turned into machine code
- execution stops when a guard fails
- after a guard failure, go back to interpreting program

Dealing With Control Flow

- an if statement in a loop is turned into a guard
- if that guard fails often, things are inefficient
- solution: attach a new trace to a guard, if it fails often enough
- new trace can lead back to same loop
- or to some other loop

Dispatching in a Tracing JIT

- trace contains bytecode operations
- bytecodes often have complex semantics
- optimizer often type-specializes the bytecodes
- according to the concrete types seen during tracing
- need to duplicate language semantics in optimizer for that

Example: Dispatching in a Tracing JIT

```
x = ADD(a : Integer, b : Integer)
```

Example: Dispatching in a Tracing JIT

```
guard_class(a, Integer)
guard_class(b, Integer)
u_a = unbox(a)
u_b = unbox(b)
u_x = int_add(a, b)
x = new(Integer, u_x)
```

Dispatching Dependencies in a Tracing JIT

- one consequence of the tracing approach:
- paths are split aggressively
- control flow merging happens at beginning of loop only
- after a type check, the rest of the trace can assume that type
- only deal with paths that are actually seen

Example: Dependencies in a Tracing JIT

```
guard_class(a, Integer)
guard_class(b, Integer)
u_a = unbox(a)
u_b = unbox(b)
u_x = int_add(u_a, u_b)
x = new(Integer, u_x)

guard_class(x, Integer)
guard_class(c, Integer)
u_x2 = unbox(x)
u_c = unbox(c)
u_r = int_add(u_x2, u_c)
r = new(Integer, u_r)
```

Example: Dependencies in a Tracing JIT

```
guard_class(a, Integer)
guard_class(b, Integer)
u_a = unbox(a)
u_b = unbox(b)
u_x = int_add(u_a, u_b)
x = new(Integer, u_x)
```

~~guard_class(x, Integer)~~

```
guard_class(c, Integer)
u_x2 = unbox(x)
u_c = unbox(c)
u_r = int_add(u_x2, u_c)
r = new(Integer, u_r)
```

Boxing Optimizations in a Tracing JIT

- possibility to do escape analysis within the trace
- only optimize common path
- i.e. the one where the object doesn't escape

Example: Boxing in a Tracing JIT

```
guard_class(a, Integer)
guard_class(b, Integer)
u_a = unbox(a)
u_b = unbox(b)
u_x = int_add(u_a, u_b)
x = new(Integer, u_x)

guard_class(c, Integer)
u_x2 = unbox(x)
u_c = unbox(c)
u_r = int_add(u_x2, u_c)
r = new(Integer, u_r)
```

Example: Boxing in a Tracing JIT

```
guard_class(a, Integer)
guard_class(b, Integer)
u_a = unbox(a)
u_b = unbox(b)
u_x = int_add(u_a, u_b)
x = new(Integer, u_x)

guard_class(c, Integer)
u_x2 = unbox(x)
u_c = unbox(c)
u_r = int_add(u_x, u_c)
r = new(Integer, u_r)
```

Advantages of Tracing JITs

- can be added to an existing interpreter unobtrusively
- interpreter does most of the work
- automatic inlining
- deals well with finding the few common paths through the large space

Bad Points of the Approach

- switching between interpretation and machine code execution takes time
- problems with really complex control flow
- granularity issues: often interpreter bytecode is too coarse
- if this is the case, the optimizer needs to carefully re-add the decision tree

How do Tracing JITs Meet the Requirements?

	Interpreter	Static Compiler	Method Compiler	**Tracing JIT**	OO VMs
Control Flow	-	+	+	+	
Late Binding	-	-	+	+	
Dispatching	-	-	+	+	
Dependencies	-	-	?	++	
Boxing	-	-	?	+	
(Reified Frames)	-	--	?	+	

Implementing Languages on Top of OO VMs

- approach: implement on top of the JVM or the CLR
- usually by compiling to the target bytecode
- plus an object model implementation
- brings its own set of benefits of problems

Implementing Languages on Top of OO VMs

- approach: implement on top of the JVM or the CLR
- usually by compiling to the target bytecode
- plus an object model implementation
- brings its own set of benefits of problems

Python Case

- Jython is a Python-to-Java-bytecode compiler
- IronPython is a Python-to-CLR-bytecode compiler

Benefits of Implementing on Top of OO VMs

- higher level of implementation
- the VM supplies a GC and a JIT
- better interoperability than what the C level provides

Benefits of Implementing on Top of OO VMs

- higher level of implementation
- the VM supplies a GC and a JIT
- better interoperability than what the C level provides

Python Case

- both Jython and IronPython integrate well with their host OO VM
- both have proper threading

The Problems of OO VMs

- often hard to map concepts of the dynamic language
- performance not improved because of the semantic mismatch
- untypical code in most object models
- object model typically has many megamorphic call sites

The Problems of OO VMs

- often hard to map concepts of the dynamic language
- performance not improved because of the semantic mismatch
- untypical code in most object models
- object model typically has many megamorphic call sites
- escape analysis cannot help with boxing, due to escaping paths
- to improve, very careful manual tuning is needed
- VM does not provide enough customization/feedback

Examples of Problems

- both Jython and IronPython are quite a bit slower than CPython
- IronPython misses reified frames

Examples of Problems

- both Jython and IronPython are quite a bit slower than CPython
- IronPython misses reified frames
- for languages like Prolog it is even harder to map the concepts

The Future of OO VMs?

- the problems described might improve in the future
- JVM will add extra support for more languages
- i.e. tail calls, `InvokeDynamic`, ...
- has not really landed yet
- good performance needs a huge amount of tweaking
- controlling the VM's behaviour is brittle:
  VMs not meant for people who care about exact shape of assembler

The Future of OO VMs?

- the problems described might improve in the future
- JVM will add extra support for more languages
- i.e. tail calls, `InvokeDynamic`, ...
- has not really landed yet
- good performance needs a huge amount of tweaking
- controlling the VM's behaviour is brittle:
  VMs not meant for people who care about exact shape of assembler

Ruby Case

- JRuby tries really hard to be a very good implementations
- took an enormous amount of effort
- tweaking is essentially Hotspot-specific

How do OO VMs Meet the Requirements?

	Interpreter	Static Compiler	Method Compiler	Tracing JIT	**OO VMs**
Control Flow	-	+	+	+	+
Late Binding	-	-	+	+	+
Dispatching	-	-	+	+	+
Dependencies	-	-	?	++	?
Boxing	-	-	?	+	?
(Reified Frames)	-	--	?	+	?

The PyPy Project

- started in 2003, received funding from the EU, Google, Nokia and some smaller companies
- goal: "The PyPy project aims at producing a flexible and fast Python implementation."
- technology should be reusable for other dynamic languages

The PyPy Project

- started in 2003, received funding from the EU, Google, Nokia and some smaller companies
- goal: "The PyPy project aims at producing a flexible and fast Python implementation."
- technology should be reusable for other dynamic languages

Language Status

- the fastest Python implementation, very complete
- contains a reasonably good Prolog
- full Squeak, but no JIT for that yet
- various smaller experiments (JavaScript, Scheme, Haskell)

PyPy's Approach to VM Construction

Goal: achieve flexibility, simplicity and performance together

- Approach: auto-generate VMs from high-level descriptions of the language
- ... using meta-programming techniques and aspects
- high-level description: an interpreter written in a high-level language
- ... which we translate (i.e. compile) to a VM running in various target environments, like C/Posix

PyPy's Approach to VM Construction

Goal: achieve flexibility, simplicity and performance together

- Approach: auto-generate VMs from high-level descriptions of the language
- ... using meta-programming techniques and aspects
- high-level description: an interpreter written in a high-level language
- ... which we translate (i.e. compile) to a VM running in various target environments, like C/Posix, CLR, JVM

PyPy

- PyPy = Python interpreter written in RPython + translation toolchain for RPython

PyPy

- PyPy = Python interpreter written in RPython + translation toolchain for RPython

What is RPython

- RPython is a (large) subset of Python
- subset chosen in such a way that type-inference can be performed
- still a high-level language (unlike SLang or PreScheme)

Auto-generating VMs

- we need a custom translation toolchain to compile the interpreter to a full VM
- many aspects of the final VM are orthogonal from the interpreter source: they are inserted during translation

Auto-generating VMs

- we need a custom translation toolchain to compile the interpreter to a full VM
- many aspects of the final VM are orthogonal from the interpreter source: they are inserted during translation

Examples

- Garbage Collection strategy
- non-trivial translation aspect: auto-generating a tracing JIT compiler from the interpreter

Good Points of the Approach

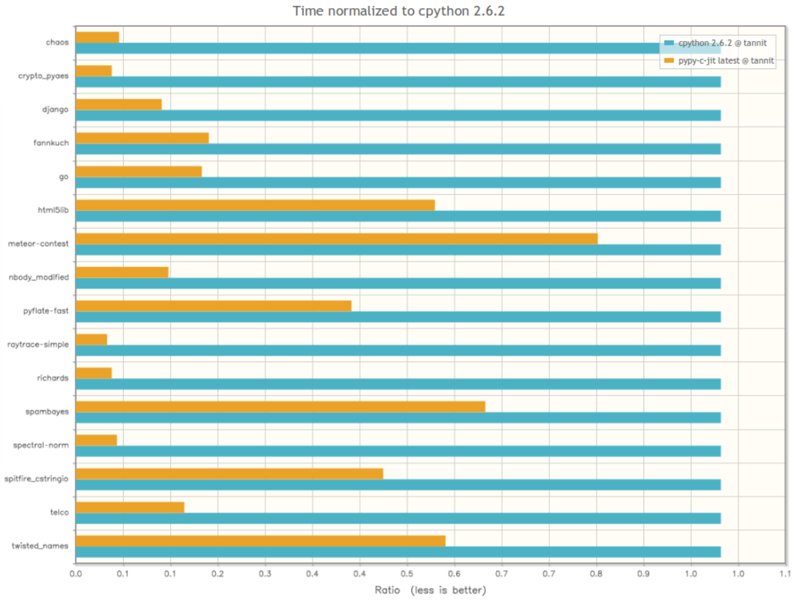
Simplicity: separation of language semantics from low-level details

Good Points of the Approach

Simplicity: separation of language semantics from low-level details
Flexibility high-level implementation language eases things (meta-programming)

Good Points of the Approach

Simplicity: separation of language semantics from low-level details
Flexibility high-level implementation language eases things (meta-programming)
Performance: "reasonable" baseline performance, can be very good with JIT


Time normalized to cpython 2.6.2
cpython 2.6.2 @ tannit
pypy-c-jit latest @ tannit
chaos
crypto_pyaes
django
fannkuch
go
html5lib
meteor-contest
nbody_modified
pyflate-fast
raytrace-simple
richards
spambayes
spectral-norm
spitfire_cstringio
telco
twisted_names
0.0
0.1
0.1
0.2
0.3
0.4
0.4
0.5
0.6
0.7
0.7
0.8
0.9
1.0
1.0
1.1
Ratio (less is better)

Meta-Tracing

Problems of Tracing JITs:

- specific to one language's bytecode
- bytecode has wrong granularity
- internals of an operation not visible in trace

Meta-Tracing

Problems of Tracing JITs:

- specific to one language's bytecode
- bytecode has wrong granularity
- internals of an operation not visible in trace

PyPy's Idea:

- write interpreters in RPython
- trace the execution of the RPython code
- using one generic RPython tracer
- the process is customized via hints in the interpreter
- no language-specific bugs

Interpreter Overhead

- most immediate problem with meta-tracing
- interpreter typically has a bytecode dispatch loop
- not a good idea to trace that

Interpreter Overhead

- most immediate problem with meta-tracing
- interpreter typically has a bytecode dispatch loop
- not a good idea to trace that
- solved by a simple trick:
- unroll the bytecode dispatch loop
- control flow then taken care of

Optimizing Late Binding and Dispatching

- late binding and dispatching code in the interpreter is traced
- as in a normal tracing JIT, the meta-tracer is good at picking common paths
- a number of hints to fine-tune the process

Optimizing Boxing Overhead

- boxing optimized by a powerful general optimization on traces
- tries to defer allocations for as long as possible
- allocations only happen in those (rare) paths where they are needed

Optimizing Boxing Overhead

- boxing optimized by a powerful general optimization on traces
- tries to defer allocations for as long as possible
- allocations only happen in those (rare) paths where they are needed

Use Cases

- arithmetic
- argument holder objects
- frames of inlined functions

Dealing With Reified Frames

- interpreter needs a frame object to store its data anyway
- those frame objects are specially marked
- JIT special-cases them
- their attributes can live in CPU registers/stack
- on reflective access, machine code is left, interpreter continues

Dealing With Reified Frames

- interpreter needs a frame object to store its data anyway
- those frame objects are specially marked
- JIT special-cases them
- their attributes can live in CPU registers/stack
- on reflective access, machine code is left, interpreter continues
- nothing deep, but a lot of engineering

Feedback from the VM

- in the beginning the hints are often not optimal yet
- to understand how to improve them, the traces must be read
- traces are in a machine-level intermediate representation
- not machine code
- corresponds quite closely to RPython interpreter code
- visualization and profiling tools

Drawbacks / Open Issues / Further Work

- writing the translation toolchain in the first place takes lots of effort (but it can be reused)
- writing a good GC was still necessary, not perfect yet
- dynamic compiler generation seems to work now, but took <u>very</u> long to get right
- granularity of tracing is sometimes not optimal, very low level

Conclusion

- PyPy solves many of the problems of dynamic language implementations
- it uses a high-level language
  - to ease implementation
  - for better analyzability
- it gives good feedback to the language implementor
- and provides various mechanisms to express deeply different language semantics

Conclusion

- PyPy solves many of the problems of dynamic language implementations
- it uses a high-level language
    - to ease implementation
    - for better analyzability
- it gives good feedback to the language implementor
- and provides various mechanisms to express deeply different language semantics
- only one solution in this design space (SPUR is another)
- more experiments needed