

# Tracing the Meta-Level: PyPy's Tracing JIT Compiler

Carl Friedrich Bolz, Antonio Cuni,  
Maciej Fijalkowski, Armin Rigo

ICOOOLPS '09 Genova, Italy

# Introduction

- ▶ increasing popularity for Dynamic Languages
- ▶ major drawback: slower than statically types languages
- ▶ most implementations without advanced techniques like JITs
- ▶ interpreters are easy, just-in-time compilation is hard

# Introduction

## What We want to do

- ▶ improve performance of interpreters written with the PyPy toolchain
- ▶ by using a tracing JIT compiler
- ▶ but trace the interpreter execution instead of the user program

# The PyPy Project

## What is PyPy?

- ▶ environment for writing flexible implementations of dynamic languages
- ▶ implementations are written in RPython
  - ▶ subset of Python that allows type inference
- ▶ language interpreter can be translated to various target environments

# The PyPy project

## Advantages for writing VMs in High-Level Languages

- ▶ free of low-level details
  - ▶ memory management
  - ▶ threading model
  - ▶ object layout
- ▶ features are added automatically during translation process

# The PyPy project

## The Translation Process

1. control flow graph construction and type inference
2. several steps to transform the intermediate representation into final executable:
  - ▶ first: make details of python object model explicit in intermediate representation
  - ▶ later: introduce garbage collection and other low-level details

# Tracing JIT compilers

- ▶ initially explored by the Dynamo Project
- ▶ same techniques were used to implement JIT compiler for JVM
- ▶ turn out to be a relatively easy way to implement JIT compilers for dynamic languages
- ▶ used for Mozilla's TraceMonkey JavaScript VM and Adobe's Tamarin ActionScript VM

# Tracing JIT compilers

## Assumptions for building Tracing JITs

- ▶ programs spend most of their time in loops
- ▶ several iterations (of one loop) take similar code paths

# Tracing JIT compilers

What should be traced?

- ▶ only generate machine code for hot code paths of commonly executed loops
- ▶ interpret the rest

# Tracing JIT compilers

## Tracing JIT Phases I

- ▶ interpretation/profiling
  - ▶ establish frequently run loops by counting backward jump instructions
- ▶ tracing
  - ▶ record history of all operations executed during one iteration of a hot loop
- ▶ code generation
  - ▶ use trace to generate efficient machine code
- ▶ code execution
  - ▶ use generated code in the next iteration

# Tracing JIT compilers

## Tracing JIT Phases II

- ▶ guards
  - ▶ ensure correctness
  - ▶ place guards at every point where the code could take another direction
  - ▶ failing a guard will return to interpretation
- ▶ position key
  - ▶ used to recognize the corresponding loop for a trace
  - ▶ describes position of the executed program
  - ▶ contains currently executed function and program counter of tracing interpreter
  - ▶ check position key at backward branch instruction (or other position changing instructions)

# Tracing JIT compilers

## Example for a Trace

```
def f(a, b):
    if b % 46 == 41:
        return a - b
    else:
        return a + b
def strange_sum(n):
    result = 0
    while n >= 0:
        result = f(result, n)
        n -= 1
    return result

# corresponding trace:
loop_header(result0, n0)
i0 = int_mod(n0, Const(46))
i1 = int_eq(i0, Const(41))
guard_false(i1)
result1 = int_add(result0, n0)
n1 = int_sub(n0, Const(1))
i2 = int_ge(n1, Const(0))
guard_true(i2)
jump(result1, n1)
```

# Applying a Tracing JIT to an Interpreter

## Terminology

- ▶ *tracing interpreter*: interpreter the JIT uses to perform tracing
- ▶ *language interpreter*: runs the user program
- ▶ *user program*: program that the language interpreter executes (program that is being executed using the VM)
- ▶ *interpreter loops*: loops inside the language interpreter
- ▶ *user loops*: loops inside the user program

# Applying a Tracing JIT to an Interpreter

## Tracing the Language Interpreter Loop

- ▶ tracing jit is not applied to the user program but to the interpreter running that program
- ▶ problem:
  - ▶ only hot loop: bytecode dispatching
  - ▶ each iteration usually interprets different bytecodes
  - ▶ in contrast to the assumption that a hot loop take similar code paths

# Applying a Tracing JIT to an Interpreter

## Tracing the Language Interpreter Loop Example

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                pc = target
        elif opcode == MOV_A_R:
            n = ord(bytecode[pc])
            pc += 1
            regs[n] = a
        [...]
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a
```

```
loop_start(a0, regs0, bytecode0,
opcode0 = strgetitem(bytecode0,
pc1 = int_add(pc0, Const(1))
guard_value(opcode0, Const(7))
a1 = int_sub(a0, Const(1))
jump(a1, regs0, bytecode0, pc1)
```

# Applying a Tracing JIT to an Interpreter

## How to Trace the User Program

- ▶ do not trace a single opcode, but a series of several opcodes by unrolling the bytecode dispatch loop
- ▶ unroll in such a way that the trace corresponds to a user loop

# Applying a Tracing JIT to an Interpreter

## Detecting a User Loop

- ▶ user loops occur when the program counter of the language interpreter has the same value several times
- ▶ in this case the program counter is represented by the variables pc and bytecode
- ▶ for the jit to know which variables are the program counter the author needs to place hints
- ▶ the jit will add these variables to the position key
- ▶ this way the jit will consider the loop to be closed if these variables are the same a second time (-> user loop)

# Applying a Tracing JIT to an Interpreter

## When to check for a Closing Loop

- ▶ program counter of language interpreter can only be the same a second time if it is set to an earlier version by an instruction
- ▶ this only happens at a backward jump in the language interpreter
- ▶ jit needs another hint to recognize a backward jump

# Applying a Tracing JIT to an Interpreter

## When to use Generated Code

- ▶ problem: all pieces of assembler code correspond to the same hot loop, the bytecode dispatch loop
- ▶ however: they correspond to different paths through that loop
- ▶ need to check the program counter to pick the correct machine code
- ▶ if program counter corresponds to the position key of one of the machine codes, this code can be executed
- ▶ this only needs to be checked at a backward branch

# Applying a Tracing JIT to an Interpreter

## Applying Hints

- ▶ green variables: should be considered as part of the program counter by the jit
- ▶ red variables: not important for the program counter
- ▶ `jit_merge_point`: put at the beginning of the bytecode dispatch loop
- ▶ `can_enter_jit`: marks the backward jump

# Applying a Tracing JIT to an Interpreter

## Applying Hints Example

```
def interpret(bytecode, a):
    regs = [0] * 256
    pc = 0
    while True:
        tlrjitdriver.jit_merge_point()
        opcode = ord(bytecode[pc])
        pc += 1
        if opcode == JUMP_IF_A:
            target = ord(bytecode[pc])
            pc += 1
            if a:
                if target < pc:
                    tlrjitdriver.can_enter_jit()
                pc = target
        elif opcode == MOV_A_R:
            ...# rest unmodified
```

# Improving the Result I

- ▶ result of the trace is not optimized enough
- ▶ most operations are not part of the user program
- ▶ a lot of operations needed to manipulate the interpreter
  - ▶ read bytestring
  - ▶ manipulate program counter

## Improving the Result II

- ▶ if operations are side effect free these computations can be folded away
- ▶ which is the case as the bytecode string is immutable
- ▶ see paper page 22

# Implementation Issues I

- ▶ JIT should not be integrated, but added during C translation
- ▶ interpretation of language interpreter is bad → double-interpretation overhead
- ▶ instead run language interpreter as a C program
- ▶ on finding a hot loop in user program start tracing
- ▶ use current state to trace bytecode representation
  - ▶ needs machine- and bytecode version in executable of VM

## Implementation issues II

- ▶ on failing a guard, fall back to normale interpretation
- ▶ problem: hard to rebuild state in the middle of execution
  - ▶ fallback deep in the code needs a deep C stack
- ▶ solution: use fallback interpreter
  - ▶ variant of language interpreter without tracing
  - ▶ run until a safe point is reached where it is easy to resume operation in C version
  - ▶ happens after at most one bytecode operation

# Evaluation

## Small Interpreter

		Time (ms)	speedup
1	Compiled to C, no JIT	$442.7 \pm 3.4$	1.00
2	Normal Trace Compilation	$1518.7 \pm 7.2$	0.29
3	Unrolling of Interp. Loop	$737.6 \pm 7.9$	0.60
4	JIT, Full Optimizations	$156.2 \pm 3.8$	2.83
5	Profile Overhead	$515.0 \pm 7.2$	0.86

- ▶ 2: slower than normaler interpretation due to overhead
- ▶ 3: better than Benchmark2, but still slower than normal interpretation
- ▶ 4: almost thee times faster
- ▶ 5: profiling overhead almost 20% of speed

# Evaluation

## PyPy

```
def f(a):  
    t = (1, 2, 3)  
    i = 0  
    while i < a:  
        t = (t[1], t[2], t[0])  
        i += t[0]  
    return i
```

		Time (s)	speedup
1	PyPy compiled to C, no JIT	23.44 ± 0.07	1.00
2	PyPy comp'd to C, with JIT	3.58 ± 0.05	6.54
3	CPython 2.5.2	4.96 ± 0.05	4.73
4	CPython 2.5.2 + Psyco 1.6	1.51 ± 0.05	15.57

- ▶ PyPy with JIT is faster than CPython, but slower than CPython with Psyco

# Related Work

## DynamoRio

- ▶ applying a trace-based optimizer to an interpreter
- ▶ adding hints helping the tracer produce better results
- ▶ same unrolling as in this paper
- ▶ however: tracing on machine code level
  - ▶ no high-level information
  - ▶ need much more hints
  - ▶ advanced optimization techniques not possible

# Related Work

Piumarta and Riccardi

- ▶ copy fragments of commonly occurring bytecode sequences together
- ▶ reduces dispatch overhead
- ▶ problem: dispatching still needed to jump between those fragments and for non-copyable bytecodes

# Related Work

Ertl and Gregg

- ▶ get rid of all dispatch overhead
- ▶ stitch together concatenated sequences by patching copied machine code
- ▶ won't help if bytecode itself does a lot of work and dispatch overhead is low

# Related Work

## Partial Evaluation

- ▶ some arguments are known (static)
- ▶ some unknown (dynamic)
- ▶ constant-fold all static arguments

# Related Work

## Partial Evaluation at Runtime

- ▶ Tempo for C
  - ▶ normal partial evaluator "packaged as library"
  - ▶ specializations are pre-determined
- ▶ DyC
- ▶ Problem: targeting the C language makes higher-level specialization difficult

# Related Work

## Dynamic Partial Evaluation

- ▶ There have been some attempts to do dynamic partial evaluation, which is partial evaluation that defers partial evaluation completely to runtime to make partial evaluation more useful for dynamic languages.
- ▶ WTF!?

# Conclusion and Next Steps

- ▶ Allocation Removal
  - ▶ objects allocated inside the loop and do not escape the loop don't need to be allocated on the heap
- ▶ Optimizing Frame Objects
  - ▶ reflection of dynamic languages allow introspection of the frame object
  - ▶ thus storing intermediate result into the frame object rendering allocation removal ineffective
  - ▶ solution: update frame object lazily