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

Carl Friedrich Bolz[1] Antonio Cuni[2] Maciej Fijałkowski[3]
Armin Rigo

[1]Softwaretechnik und Programmiersprachen
Heinrich-Heine-Universität Düsseldorf
[2]University of Genova, Italy
[3]merlinux GmbH

6th of July 2009, ICOOOLPS '09, Genova

Motivation

- writing good JIT compilers for dynamic programming languages is hard and error-prone
- tracing JIT compilers are a new approach to JITs that are supposed to be easier
- what happens when a tracing JIT is applied “one level down”, i.e. to an interpreter
- how to solve the occurring problems

Context: The PyPy Project

- a general environment for implementing dynamic languages
- contains a compiler for a subset of Python (“RPython”)
- interpreters for dynamic languages written in that subset
- various interpreters written with PyPy: Python, Prolog, Smalltalk, Scheme, JavaScript, GameBoy emulator
- can be translated to a variety of target environment: C, JVM, .NET

Tracing JIT Compilers

- idea from Dynamo project: dynamic rewriting of machine code
- later used for a lightweight Java JIT
- seems to also work for dynamic languages (see TraceMonkey)

Tracing JIT Compilers

- idea from Dynamo project: dynamic rewriting of machine code
- later used for a lightweight Java JIT
- seems to also work for dynamic languages (see TraceMonkey)

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 JIT Compilers

- mixed-mode execuction 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
- when a full loop is traced, the trace is converted to machine code

Example

```
def strange_sum(n):
    result = 0
    while n >= 0:
        result = f(result, n)
        n -= 1
    return result

def f(a, b):
    if b % 46 == 41:
        return a - b
    else:
        return a + b
```

Example

```
def strange_sum(n):
    result = 0
    while n >= 0:
        result = f(result, n)
        n -= 1
    return result

def f(a, b):
    if b % 46 == 41:
        return a - b
    else:
        return a + b

# 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)
```

(Dis-)Advantages of Tracing JITs

Good Points of the Approach

- easy and fast machine code generation: needs so support only one path
- (things are more complex, but let's ignore that for now)
- interpreter does a lot of the work
- can be added to an existing interpreter unobtrusively
- automatic inlining
- produces comparatively little machine code

(Dis-)Advantages of Tracing JITs

Good Points of the Approach

- easy and fast machine code generation: needs so support only one path
- (things are more complex, but let's ignore that for now)
- interpreter does a lot of the work
- can be added to an existing interpreter unobtrusively
- automatic inlining
- produces comparatively little machine code

Bad Points of the Approach

- unclear whether assumptions are true often enough
- switching between interpretation and machine code execution takes time

Applying a Tracing JIT to an Interpreter

- Question: What happens if the program is itself a bytecode interpreter?
- the (most important) hot loop of a bytecode interpreter is the bytecode dispatch loop
- Assumption violated: consecutive iterations of the dispatch loop will usually take very different code paths
- what can we do?

Applying a Tracing JIT to an Interpreter

- Question: What happens if the program is itself a bytecode interpreter?
- the (most important) hot loop of a bytecode interpreter is the bytecode dispatch loop
- Assumption violated: consecutive iterations of the dispatch loop will usually take very different code paths
- what can we do?

Terminology

- *tracing interpreter:* the interpreter that originally runs the program and produces traces
- *language interpreter:* the bytecode interpreter run on top
- *user program:* the program run by the language interpreter

```
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_R_A:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == MOV_A_R:
            ...
        elif opcode == ADD_R_TO_A:
            ...
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a
```

```
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_R_A:
            n = ord(bytecode[pc])
            pc += 1
            a = regs[n]
        elif opcode == MOV_A_R:
            ...
        elif opcode == ADD_R_TO_A:
            ...
        elif opcode == DECR_A:
            a -= 1
        elif opcode == RETURN_A:
            return a
```

```
# Example bytecode
# Square the accumulator:

MOV_A_R     0  # i = a
MOV_A_R     1  # copy of 'a'

# 4:
MOV_R_A     0  # i--
DECR_A
MOV_A_R     0

MOV_R_A     2  # res += a
ADD_R_TO_A  1
MOV_A_R     2

MOV_R_A     0  # if i!=0:
JUMP_IF_A   4  #    goto 4

MOV_R_A     2  # return res
RETURN_A
```

Trace

Resulting trace when tracing bytecode `DECR_A`:

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

Idea for a Solution

- goal: try to trace the loops *in the user program,* and not just one iteration of the bytecode dispatch loop
- tracing interpreter needs information about the language interpreter
- provided by adding *three hints* to the language interpreter

Idea for a Solution

- goal: try to trace the loops *in the user program,* and not just one iteration of the bytecode dispatch loop
- tracing interpreter needs information about the language interpreter
- provided by adding *three hints* to the language interpreter

Hints Give Information About:

- which variables make up the program counter of the language interpreter (together those are called *position key*)
- where the bytecode dispatch loop is
- which bytecodes can correspond to backward jumps

Interpreter with Hints

```
tlrjitdriver = JitDriver(['pc', 'bytecode'])

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

Modifying Tracing

- goal: try to trace the loops *in the user program,* and not just one iteration of the bytecode dispatch loop
- tracing interpreter stops tracing only when:
  - it sees a backward jump in the language interpreter
  - the position key of the language interpreter matches an ealier value
- in this way, full user loops are traced

Result When Tracing SQUARE

```
loop_start(a0, regs0, bytecode0, pc0)
# MOV_R_A 0
opcode0 = strgetitem(bytecode0, pc0)
pc1 = int_add(pc0, Const(1))
guard_value(opcode0, Const(2))
n1 = strgetitem(bytecode0, pc1)
pc2 = int_add(pc1, Const(1))
a1 = list_getitem(regs0, n1)
# DECR_A
# MOV_A_R 0
# MOV_R_A 2
# ADD_R_TO_A 1
# MOV_A_R 2
# MOV_R_A 0
...
# JUMP_IF_A 4
opcode6 = strgetitem(bytecode0, pc13)
pc14 = int_add(pc13, Const(1))
guard_value(opcode6, Const(3))
target0 = strgetitem(bytecode0, pc14)
pc15 = int_add(pc14, Const(1))
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0, bytecode0, target0)
```

What Have We Won?

- trace corresponds to one loop of the user program
- however, most operations are concerned with manipulating bytecode and program counter
- bytecode and program counter are part of the position key
- thus they are constant at the beginning of the loop
- therefore they can and should be constant-folded

Result When Tracing `SQUARE` With Constant-Folding

```
loop_start(a0, regs0)
# MOV_R_A 0
a1 = list_getitem(regs0, Const(0))
# DECR_A
a2 = int_sub(a1, Const(1))
# MOV_A_R 0
list_setitem(regs0, Const(0), a2)
# MOV_R_A 2
list_getitem(regs0, Const(2))
# ADD_R_TO_A  1
i0 = list_getitem(regs0, Const(1))
a4 = int_add(a3, i0)
# MOV_A_R 2
list_setitem(regs0, Const(2), a4)
# MOV_R_A 0
a5 = list_getitem(regs0, Const(0))
# JUMP_IF_A 4
i1 = int_is_true(a5)
guard_true(i1)
jump(a5, regs0)
```

Results

- almost only computations related to the user program remain
- list of registers is only vestige of language interpreter

Results

- almost only computations related to the user program remain
- list of registers is only vestige of language interpreter

Timing Results Computing Square of 10'000'000

		Time (ms)	speedup
1	No JIT	442.7 ± 3.4	1.00
2	JIT, Normal Trace Compilation	1518.7 ± 7.2	0.29
3	JIT, Unrolling of Interp. Loop	737.6 ± 7.9	0.60
4	JIT, Full Optimizations	156.2 ± 3.8	2.83

Scaling to Large Interpreters?

- we can apply this approach to PyPy's Python interpreter (70 KLOC)
- speed-ups promising (see next slide)
- no Python-specific bugs!

Timings for Python Interpreter

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

Timings of `f(10000000)`

		Time (ms)	speedup
1	PyPy compiled to C, no JIT	1793 $\pm$ 11	1.00
2	PyPy comp'd to C, with JIT	483 $\pm$ 6	3.71
3	CPython 2.6	1869 $\pm$ 11	0.96
4	CPython 2.6 + Psyco 1.6	511 $\pm$ 7	3.51

Conclusions

- some small changes to a tracing JIT makes it possible to effectively apply it to bytecode interpreters
- result is similar to a tracing JIT for that language
- bears resemblance to partial evaluation, arrived at by different means
- maybe enough to write exactly one tracing JIT?

Conclusions

- some small changes to a tracing JIT makes it possible to effectively apply it to bytecode interpreters
- result is similar to a tracing JIT for that language
- bears resemblance to partial evaluation, arrived at by different means
- maybe enough to write exactly one tracing JIT?

Outlook

- better optimizations of the traces
- escape analysis
- optimize frame objects
- speed up tracing itself
- apply to other interpreters and larger programs

Thank you! Questions?

- some small changes to a tracing JIT makes it possible to effectively apply it to bytecode interpreters
- result is similar to a tracing JIT for that language
- bears resemblance to partial evaluation, arrived at by different means
- maybe enough to write exactly one tracing JIT?

Outlook

- better optimizations of the traces
- escape analysis
- optimize frame objects
- speed up tracing itself
- apply to other interpreters and larger programs