Pydrofoil 分享

PyPy meta-tracing JIT

Mingzhu Yan

2025-07-02

PLCT Lab

Outline

1. 介绍

1.1 Pydrofoil

[Submitted on 6 Mar 2025]

Pydrofoil: accelerating Sail-based instruction set simulators

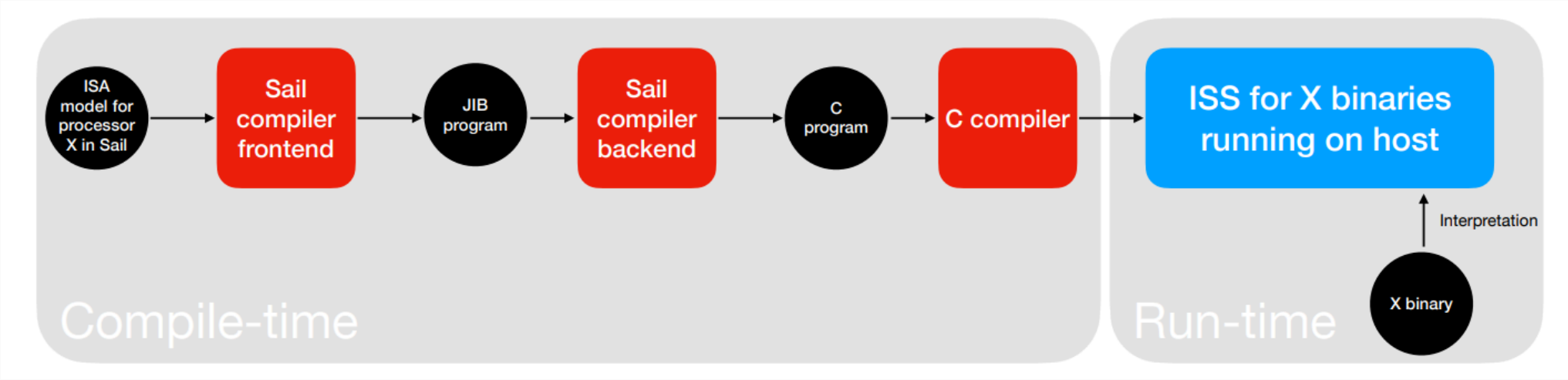
Carl Friedrich Bolz-Tereick, Luke Panayi, Ferdia McKeogh, Tom Spink, Martin Berger

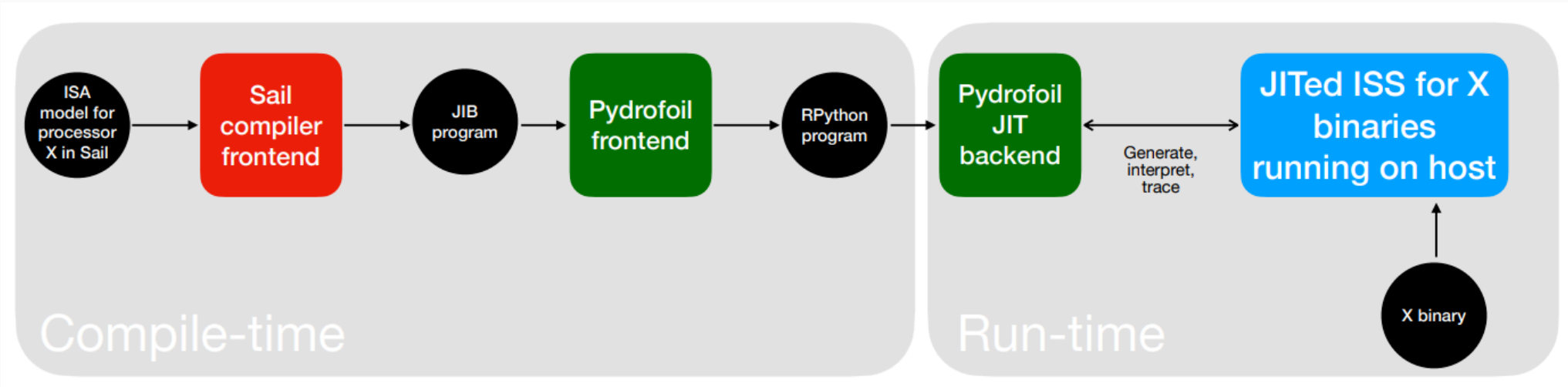
We present Pydrofoil, a multi-stage compiler that generates instruction set simulators (ISSs) from processor instruction set architectures (ISAs) expressed in the high-level, verification-oriented ISA specification language Sail. Pydrofoil shows a > 230x speedup over the C-based ISS generated by Sail on our benchmarks, and is based on the following insights. (i) An ISS is effectively an interpreter loop, and tracing just-in-time (JIT) compilers have proven effective at accelerating those, albeit mostly for dynamically typed languages. (ii) ISS workloads are highly atypical, dominated by intensive bit manipulation operations. Conventional compiler optimisations for general-purpose programming languages have limited impact for speeding up such workloads. We develop suitable domain-specific optimisations. (iii) Neither tracing JIT compilers, nor ahead-of-time (AOT) compilation alone, even with domain-specific optimisations, suffice for the generation of performant ISSs. Pydrofoil therefore implements a hybrid approach, pairing an AOT compiler with a tracing JIT built on the meta-tracing PyPy framework. AOT and JIT use domain-specific optimisations. Our benchmarks demonstrate that combining AOT and JIT compilers provides significantly greater performance gains than using either compiler alone.

```
function clause execute (ITYPE (imm, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  ⇒ rs1_val + immext,
    RISCV_SLTI  ⇒ zero_extend(bool_to_bits(rs1_val <_s immext)),
    RISCV_SLTIU ⇒ zero_extend(bool_to_bits(rs1_val <_u immext)),
    RISCV_ANDI  ⇒ rs1_val & immext,
    RISCV_ORI   ⇒ rs1_val | immext,
    RISCV_XORI  ⇒ rs1_val ^ immext
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

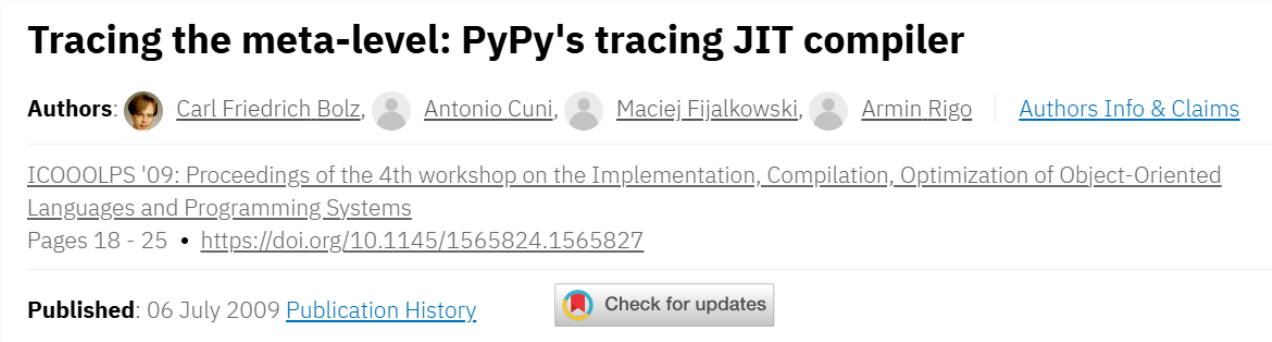
```
uint64_t z2zE10316;
{
  __label__ case_20235, case_20234, case_20233, case_20232, case

  uint64_t z3zE25035;
  {
    if ((zADDI != zuz33676)) goto case_20235;
    uint64_t z2zE10306;
    z2zE10306 = zrX_bits(zuz33674);
    z3zE25035 = ((z2zE10306 + zimmext) & UINT64_C(0×FFFFFFFF));
    goto finish_match_20229;
  }
```

1.2 Pydrofoil





1.3 PyPy meta-tracing JIT



Tracing the meta-level: PyPy's tracing JIT compiler

Authors: Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Armin Rigo | Authors Info & Claims

ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems
Pages 18 - 25 • https://doi.org/10.1145/1565824.1565827

Published: 06 July 2009 Publication History

Check for updates

- **目标** 在 PyPy 项目（一种 Python 语言实现）中针对一些动态语言（包括 Python) 的解释器程序引入 JIT 编译技术

  > 从普通解释器生成自带 *JIT* 的解释器

- **要解决的问题** tracing JIT 可以加速热点代码，但是将 tracing JIT 直接应用于一个字节码解释器程序，会导致非常有限的加速甚至完全**没有加速**

- **方法** 展开字节码调度循环引导 tracing JIT 编译器提高字节码解释器的速度

  > 需要编写一个解释器, 并且依赖作者的少量提示

2. PYPY 项目

2.1 背景

- 动态语言的发展迅速，使用广泛，但是常常 **性能较差**
  - JavaScript 正越来越多地被用于实现运行在浏览器中的全规模应用程序
  - 其他动态语言（如 Ruby、Perl、Python、PHP）则用于许多服务器领域，以及与网络无关的领域
- 对加速动态语言解释器性能的研究很多，但是 **加速技术的实际应用少**
  - 编译器固有的复杂性
  - 编写即时编译器困难
  - 动态语言的特性
- PyPy 项目在寻找一种**一般性的方法来简化动态语言的实现**

PyPy

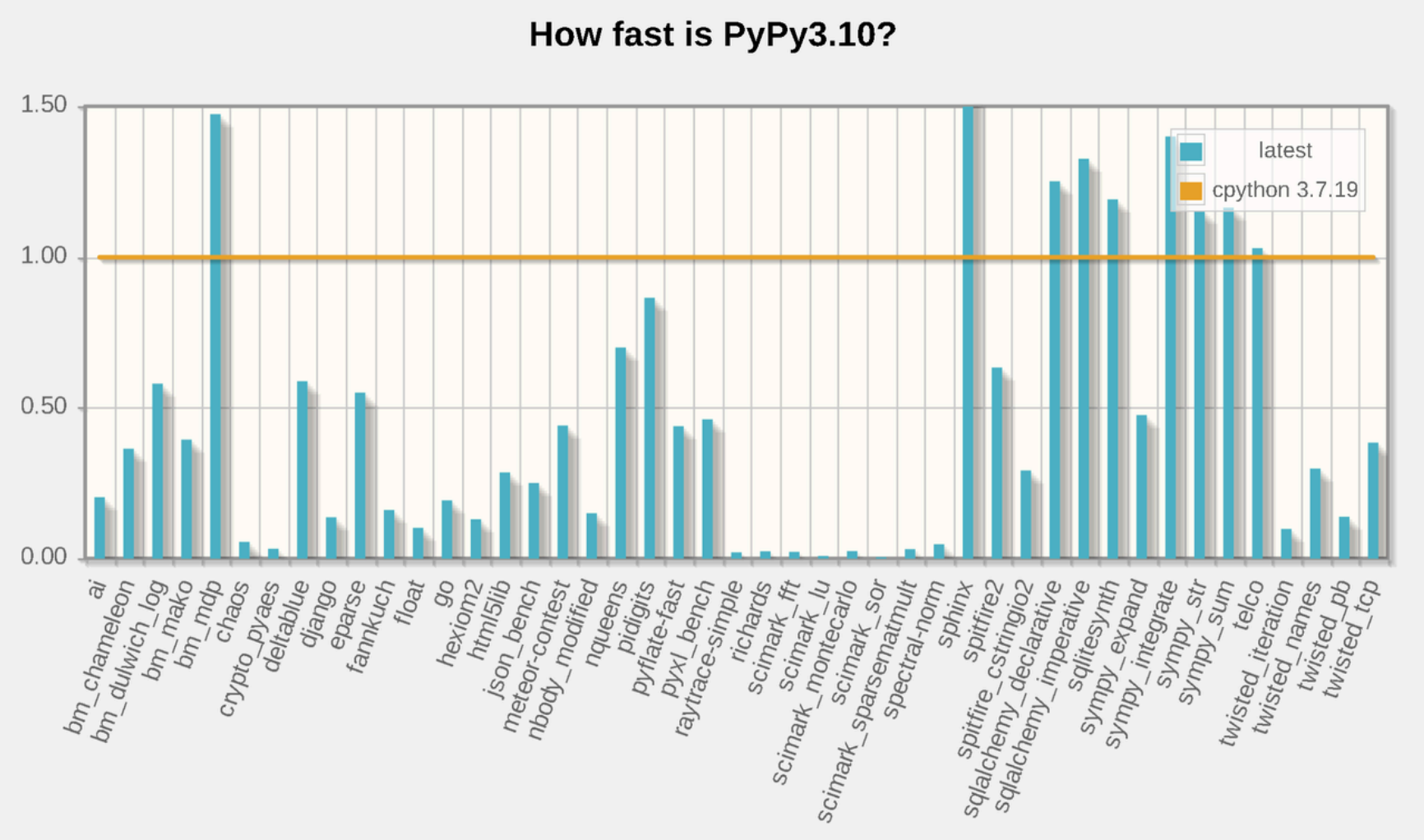
A fast, compliant alternative implementation of Python

Download PyPy

What is PyPy ?

pypy

2.2 PYPY 项目


How fast is PyPy3.10?
1.50
1.00
0.50
0.00
latest
cpython 3.7.19
ai
bm_chameleon
bm_dulwich_log
bm_mako
bm_mdp
chaos
crypto_pyaes
deltablue
django
eparse
fannkuch
float
go
hexiom2
html5lib
json_bench
meteor-contest
nbody_modified
nqueens
pidigits
pyflate-fast
pyxl_bench
raytrace-simple
richards
scimark_fft
scimark_lu
scimark_montecarlo
scimark_sor
scimark_sparsematmult
spectral-norm
sphinx
spitfire2
spitfire_cstringio2
sqlalchemy_declarative
sqlalchemy_imperative
sqlitesynth
sympy_expand
sympy_integrate
sympy_str
sympy_sum
telco
twisted_iteration
twisted_names
twisted_pb
twisted_tcp

2.3 PYPY 项目

- PyPy 项目是一个可以编写动态语言灵活实现的环境。要使用 PyPy 实现动态语言，**必须用 RPython 编写该语言的解释器**
- RPython (“Restricted Python”) 是 Python 的一个子集，特点是可以进行类型推断
  - 使用 RPython 编写的解释器可以借助 PyPy 的 translation toolchain 转换为各种目标环境，如 C/Posix、JVM 等
  - 通过用高级语言编写虚拟机，使语言的实现不受内存管理策略、线程模型或对象布局等低级细节的影响。这些特性在翻译过程中会自动添加
- 与常见的 JIT 不同，PyPy 的 tracing JIT **追踪的是解释器的执行**，而不是用户程序的执行

2.4 RPython 翻译过程

- 首先进行控制流图构建和类型推断
- 然后经过一系列中间表示转换，最终得到可执行文件
  - 第一个转换步骤使 Python 对象模型的细节转换为中间表示
  - 后续步骤引入垃圾收集和其他低级细节

> 这种程序的内部表示也用作 tracing JIT 的输入

3. TRACING JIT COMPILERS

3.1 Tracing JIT 介绍

- tracing 优化技术最初是由 Dynamo 项目探索，目的是在运行时动态优化机器代码。其技术随后成功用于实现 Java 虚拟机的 JIT 编译器
- 随后发现，tracing JIT 是一种相对简单的方法，可以为动态语言实现 JIT 编译器
- 该技术也被 Mozilla 的 TraceMonkey JavaScript 虚拟机使用，并已尝试用于 Adobe 的 Tamarin ActionScript 虚拟机

3.2 tracing JIT 特点

- tracing JIT 建立在以下基本假设之上：
  1. 程序运行的大部分时间都在循环中
  2. 同一个循环的多次迭代很可能会采取相似的代码路径
- tracing JIT 的基本方法：

  只为循环中的热点路径生成机器码，剩下的部分解释执行

  > 然而 **常见的循环已经经过了高度优化**，包括应用了激进的内联等

3.3 常规的 tracing JIT 流程

通常情况下带有 tracing JIT 的虚拟机在运行一个程序时会经过多个阶段

1. 解释执行/性能分析
   - 一开始解释执行，通过**轻量级的性能分析**统计出哪些循环执行的最频繁

   > 性能分析可以通过统计跳转指令（仅包括 backward jump）次数实现
2. 追踪热点代码
   - 当定位到热点代码后，解释器会进入 tracing mode，持续记录所有运行的指令，直到遇到热点循环
   - 被记录的信息称为 trace，它包含一系列的指令，以及实际的操作数和结果
   - 为了判断何时满足热点循环，tracing 会反复检查解释器是否处于程序中之前曾经到达过的位置
3. 代码生成和执行

3.4 tracing JIT 示例

```
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
```

- `strange_sum` 循环
- `f` 分支

3.5 trace 示例

首先是解释执行，当性能分析器发现 strange_sum 中的 while 循环被频繁执行时，tracing JIT 将开始追踪该循环的执行，形成 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)
```

3.6 将 tracing JIT 应用于解释器

PyPy 的 traceing JIT 是不寻常的，因为它不是应用于用户程序，而是应用于运行用户程序的解释器

术语

- language interpreter（使用 RPython 编写的语言解释器）
- tracing interpreter（追踪语言解释器的解释器，PyPy 框架）
- user program （用户程序）
- user loops（用户程序中的循环）
- interpreter loops (语言解释器中的解释执行循环)

（假设语言是基于字节码的）

3.7 tracing interpreter 示例

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

> 一个简单的字节码解释器（带有一些寄存器和一个 a）

对于 tracing interpreter 来说，最常见的热点循环是 **bytecode dispatch loop**

对一些简单的解释器来说， 这甚至可能是唯一的热点循环

3.8 tracing interpreter trace 示例

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

```
while true {
  a -= 1;
}
```

```
DECR_A
DECR_A
DECR_A
DECR_A
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)
```

Const(7) => DECR_A

循环分支断言分支几乎每次都成功

3.9 字节码解释执行示例

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

```
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: goto 4
JUMP_IF_A   4

MOV_R_A     2   # return res
RETURN_A
```

计算 a 的平方

```
function (a) {
  i = a;
  while i != 0 {
    i -= 1;
    res += a;
  }
  return 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)
```

循环分支断言分支几乎每次都失败

3.10 meta-tracing JIT 的优化

1. 同时追踪多个 opcode，高效地展开 dispatch loop （理想情况下，展开的大小应开等同于 user loop 的大小）
2. 当 PC 多次出现相同值，则意味着用户程序出现了循环，因为 JIT 无法知道哪些变量存储了 PC，所以用户需要标记这些变量

3.11 按照 PC 展开 user loop

```
# 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: goto 4
JUMP_IF_A    4
```

```
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
opcode1 = strgetitem(bytecode0, pc2)
pc3 = int_add(pc2, Const(1))
guard_value(opcode1, Const(7))
a2 = int_sub(a1, Const(1))
# MOV_A_R 0
opcode1 = strgetitem(bytecode0, pc3)
pc4 = int_add(pc3, Const(1))
guard_value(opcode1, Const(1))
n2 = strgetitem(bytecode0, pc4)
pc5 = int_add(pc4, Const(1))
list_setitem(regs0, n2, a2)
# MOV_R_A 2
...
# ADD_R_TO_A 1
opcode3 = strgetitem(bytecode0, pc7)
pc8 = int_add(pc7, Const(1))
guard_value(opcode3, Const(5))
n4 = strgetitem(bytecode0, pc8)
pc9 = int_add(pc8, Const(1))
i0 = list_getitem(regs0, n4)
a4 = int_add(a3, i0)
# 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)
```

不再按照 opcode 分支判断热点，而是按照 PC 的变化追踪整个循环运行的所有字节码

3.12 标记 PC

```
tlrjitdriver = JitDriver(greens = ['pc', 'bytecode'],
                         reds   = ['a', 'regs'])

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
```

对变量的分类

- greens 为 PC 相关的值
- reds 为其他

方法

- jit_merge_point 在循环的开头调用
- can_enter_jit 需要在 PC 修改为更早的值之前调用
  - 这里决定是否要展开 profiling
  - 当 PC 与之前调用 can_enter_jit() 时的 PC 相同时，则意味着循环闭合

3.13 对结果的优化

```
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
a3 = 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)
```

- 引入常量折叠，PC 和 opcode 等绿色变量通常可以折叠
- RPython 中的字符串类型时不变量，也可以折叠

4. IMPLEMENTATION ISSUES

4.1 实现遇到的问题

- 如果真的解释执行 RPython 编写的语言解释器，这种双重解释的开销将无比巨大，所以 RPython 被编译为 C 运行
- 为了识别循环，并接入 JIT，最终的可执行文件嵌入了两个版本的语言解释器
  - 一个是机器代码版本，用于首次解释执行用户程序，进行性能分析，没有性能优化
  - 另一个是字节码版本，启动 trace 模式后，Tracing 解释器为了记录下完整的操作流，而不得不去“解释执行”整个语言解释器

  所以 tracing 本身是昂贵的，它会产生双重解释的开销
- 条件守卫失败的回退可能是一个复杂的过程，如果失败的函数堆栈很深，重建语言解释器，恢复状态，并从该点重新执行很困难

5. 评估

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

Figure 8: Benchmark results of example interpreter computing the square of 10000000

```
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

6. 相关工作

6.1 相关工作

- DynamoRIO 实现了同样的解释器循环展开, 但是它的 tracing 是基于汇编的, 因此受到了诸多限制
  - 无法获得解释器的高级信息
  - 需要更多的提示信息, 汇编层级无法辨别字符串等信息
  - 无法实现分配移除等高级优化
- 将解释器片段组合在一起, 将热点机器代码序列拼接在一起, 可以大大减轻调度开销 (这里 PyPy 通过内联 + 常量折叠已经大幅优化了调度开销)
- dynamic partial evaluation

7. 总结

7.1 总结

优化结果在小型解释器上的效果很好

下一步工作

- 通过逃逸分析移除堆分配
- 优化 Frame Objects