JRuby 9000
Optimizing Above the JVM
Me

- Charles Oliver Nutter
- @headius
- Red Hat
- Based in Minneapolis, Minnesota
- Ten years working on JRuby (uff da!)
JRuby 9000

- Optimizable intermediate representation
- Mixed mode runtime (now with tiers!)
- Lazy JIT to JVM bytecode
- byte[] strings and regular expressions
Intermediate Representation

- AST to semantic representation
- Traditional compiler design
- Register machine
def foo(a, b):
    c = 1
    d = a + c
end

0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)

3 address format
def foo(a, b)
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line num(1)
6  c = 1
7  line num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
```
 optimization

 def foo(a, b)
   c = 1
   d = a + c
 end

 0  check arity(2, 0, -1)
 1  a = recv pre reqd arg(0)
 2  b = recv pre reqd arg(1)
 3  %block = recv closure
 4  thread poll
 5  line num(1)
 6  c = 1
 7  line num(2)
 8  %v_0 = call(:+, a, [c])
 9  d = copy(%v_0)
10  return(%v_0)
```
-Xir.passes=LocalOptimizationPass, DeadCodeElimination

```python
def foo(a, b):
c = 1
d = a + c
end
```

```text
0  check arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
2  b = recv_pre_reqd_arg(1)
3  %block = recv_closure
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
```
def foo(a, b):
c = 1
d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
5  line_num(1)
6  c = 1
7  line_num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b)
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check arity(2, 0, -1)
1  a = recv pre reqd arg(0)
4  thread poll
5  line num(1)
6  c = 1
7  line num(2)
8  %v_0 = call(:+, a, [c])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b)
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0 check_arity(2, 0, -1)
1 a = recv_pre_reqd_arg(0)
4 thread_poll
5 line_num(1)
6 c = 1
7 line_num(2)
8 %v_0 = call(:+, a, [c])
9 d = copy(%v_0)
10 return(%v_0)
def foo(a, b):
c = 1
d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
5  line_num(1)
6  c =
7  line_num(2)
8  %v_0 = call(:+, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
def foo(a, b):
    c = 1
    d = a + c
end

-Xir.passes=LocalOptimizationPass, DeadCodeElimination

0  check arity(2, 0, -1)
1  a = recv pre reqd arg(0)
4  thread poll
5  line num(1)
7  line num(2)
8  %v_0 = call(:+, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
check_arity(2, 0, -1)
a = recv_pre_reqd_arg(0)
thread_poll
line_num(1)
line_num(2)
%v_0 = call(:+, a, [1])
d = copy(%v_0)
return(%v_0)
0  check arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
7  line_num(2)
8  %v_0 = call(:+, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
Tiers!

- Tier 1: Simple interpreter (no passes run)
- Tier 2: Full interpreter (static optimization)
- Tier 3: Full interpreter (profiled optz)
- Tier 4: JVM bytecode
- Tiers 5+: Whatever JVM does from there
Why Not Truffle?

- Startup and memory use are worse
- No integration with other JVM langs yet
- We still want to target JVM
- It's not ready yet!
Red/black tree benchmark

- JRuby no indy
- JRuby with indy
- JRuby+Truffle
- CRuby 2.3
Recent Wins

- JITable blocks
- define_method performance
- Reduced-cost transient exceptions
Block Jitting

- JRuby 1.7 only jitted methods
- Not free-standing procs/lambdas
- Not define_method blocks
- Easier to do now with 9000's IR
- Blocks JIT as of 9.0.4.0
class Foo
  define_method :test do
    self
  end
end

loop do
  puts Benchmark.measure { 1_000_000.times { call some method } }
end
Block Jitting

Performance of define_method in loaded file

- normal method
- define_method method

ruby -e "load "bench_define_method.rb""
define_method

```ruby
define_method(:add) do |a, b|
  a + b
end

names.each do |name|
  define_method(name) { send :"do_#{name}" }
end
```

Convenient for metaprogramming, but blocks have more overhead than methods.
Optimizing define_method

- Noncapturing
  - Treat as method in compiler
  - Ignore surrounding scope
- Capturing (future work)
  - Lift read-only variables as constant
Getting Better!

![Bar Chart]

- def
- define_method w/ capture

- MRI
- JRuby 9.0.1.0
- JRuby 9.0.4.0
Low-cost Exceptions

• Backtrace cost is VERY high on JVM
  • Lots of work to construct
• Exceptions frequently ignored
  • ...or used as flow control (shame!)
• If ignored, backtrace is not needed!
Postfix Antipattern

foo rescue nil

Exception raised

StandardError rescued

Exception ignored

Result is simple expression, so exception is never visible.
csv.rb Converters

```ruby
Converters = { integer: lambda { |f| Integer(f) rescue f }, float: lambda { |f| Float(f) rescue f }, ... All trivial rescues, no traces needed.
```
Simple rescue
Improvement

Iters/second

524,475

10,700

0

150000

300000

450000

600000
Much Better!

- 524,475 Iters/second
- 10,700 Iters/second
Current Work
Inlining

- 900 pound gorilla of optimization
  - shove method/closure back to callsite
  - eliminate call frames
  - eliminate parameter passing/return
But... JVM?

- JVM will inline for us, but...
  - only if we use invokedynamic
  - and the code isn't too big
  - and there's no polymorphic code
  - and it feels like it today
def decrement_one(i):
    i - 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end
def decrement_one(i):
    i - 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end

def decrement_one(i):
    i - 1
end

i = 1_000_000
while i < 0
    if guard_same? self
        i = i - 1
    else
        i = decrement_one(i)
    end
end
Today’s Inliner

def decrement_one(i):
    i - 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end

def decrement_one(i):
    i - 1
end

i = 1_000_000
while i < 0
    if guard_same? self
        i = i - 1
    else
        i = decrement_one(i)
    end
end
def decrement_one(i):
    i -= 1
end

i = 1_000_000
while i > 0
    i = decrement_one(i)
end

def decrement_one(i):
    i -= 1
end

i = 1_000_000
while i < 0
    if guard_same? self
        i = i - 1
    else
        i = decrement_one(i)
    end
end
end
Inlining Today

- Conceptually simple
- Does not deoptimize
- Works in interpreter and JIT
- Still experimental
- Stack trace reconstitution TBD
Profiling

- You can't inline if you can't profile!
- For each call site record call info
  - Which method(s) called
  - How frequently
- Inline most frequently-called method
Profiling

code.rb

0 my_array.each do |object|
1   object.execute
2   end
Profiling

code.rb

0  my_array.each do |object|
1  object.execute
2  end

monomorphic

class Foo
  def execute
    puts "FOO"
  end
end
Profiling

code.rb

```ruby
0 my_array.each do |object|
1   object.execute
2 end
```

polymorphic

```ruby
class Foo
  def execute
    puts "FOO"
  end
end

class Bar
  def execute
    puts "BAR"
  end
end
```
Profiling

- <2% overhead (to be reduced more)
- Working* (interpreter AND JIT)
- Feeds directly into inlining

* Fragile and buggy!
Profiling

```ruby
def small_loop(i)
  k = 10
  while k > 0
    k = yield(k)
  end
  i - 1
end

def big_loop(i)
  i = 100_000
  while true
    i = small_loop(i) {
      |j| j - 1
    }
    return 0 if i < 0
  end
end

900.times {
  |i| big_loop i
}
```

Like an Array#each

May see many blocks

JVM will not inline this

hot & monomorphic
Inlining FTW!

![Bar graph showing time in seconds]
Numeric Specialization

- Everything's an object
- JVM has only references and primitives
  - Not compatible in bytecode
- Need to optimize numerics as primitive
```ruby
def looper(n)
  i = 0
  while i < n
    do_something(i)
    i += 1
  end
end
```

*Probably a Fixnum?*

Cached object

New Fixnum $i + 1$

Call with $i$
def looper(n)
    \[ i = 0 \]
    while \( i < n \)
        do_something(i)
        \[ i += 1 \]
    end
end

Specialize \( n, i \) to long

def looper(long n)
    long \[ i = 0 \]
    while \( i < n \)
        do_something(i)
        \[ i += 1 \]
    end
end

Deopt to object version if \( n \) or \( i + 1 \) is not Fixnum
Interpreter FTW!

- Deopt is much simpler with interpreter
  - Collect local vars, instruction index
  - Raise exception to bail out
  - Fire up interpreter, keep going
- Much cheaper than resuming bytecode
Current Status

- Working prototype
- No deopt
- No type guards
- No overflow check for Fixnum/Bignum
```python
def iterate(x, y):
cr = y - 0.5
ci = x
zi = 0.0
zr = 0.0
i = 0
bailout = 16.0
max_iterations = 1000

while true:
    i += 1
    temp = zr * zi
    zr2 = zr * zr
    zi2 = zi * zi
    zr = zr2 - zi2 + cr
    zi = temp + temp + ci
    return i if (zi2 + zr2 > bailout)
    return 0 if (i > max_iterations)
end
end
```
When?

- Profiling, inlining mostly need testing
- Specialization needs guards, deopt
- Probably landing in next couple months
JRuby 9.1.0.0

- Coming soon (end of Feb?)
- Ruby 2.3 support
- Some of these optimizations
- More attention to performance overall
Thank You

- @headius
- @tom_enebo
- http://jruby.org