JRuby 9000

Optimizing Above the JVM

Me

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

Ruby Challenges

- Dynamic dispatch for most things
- Dynamic possibly-mutating constants
- Fixnum to Bignum promotion
- Literals for arrays, hashes: [a, b, c].sort[1]
- Stack access via closures, bindings
- Rich inheritance model

```ruby
module SayHello
  def say_hello
    "Hello, " + to_s
  end
end

class Foo
  include SayHello

  def initialize
    @my_data = {bar: 'baz', quux: 'widget'}
  end

  def to_s
    @my_data.map do |k,v|
      "#{k} = #{v}"
    end.join(', ')
  end
end

Foo.new.say_hello # => "Hello, bar = baz, quux = widget"
```

More Challenges

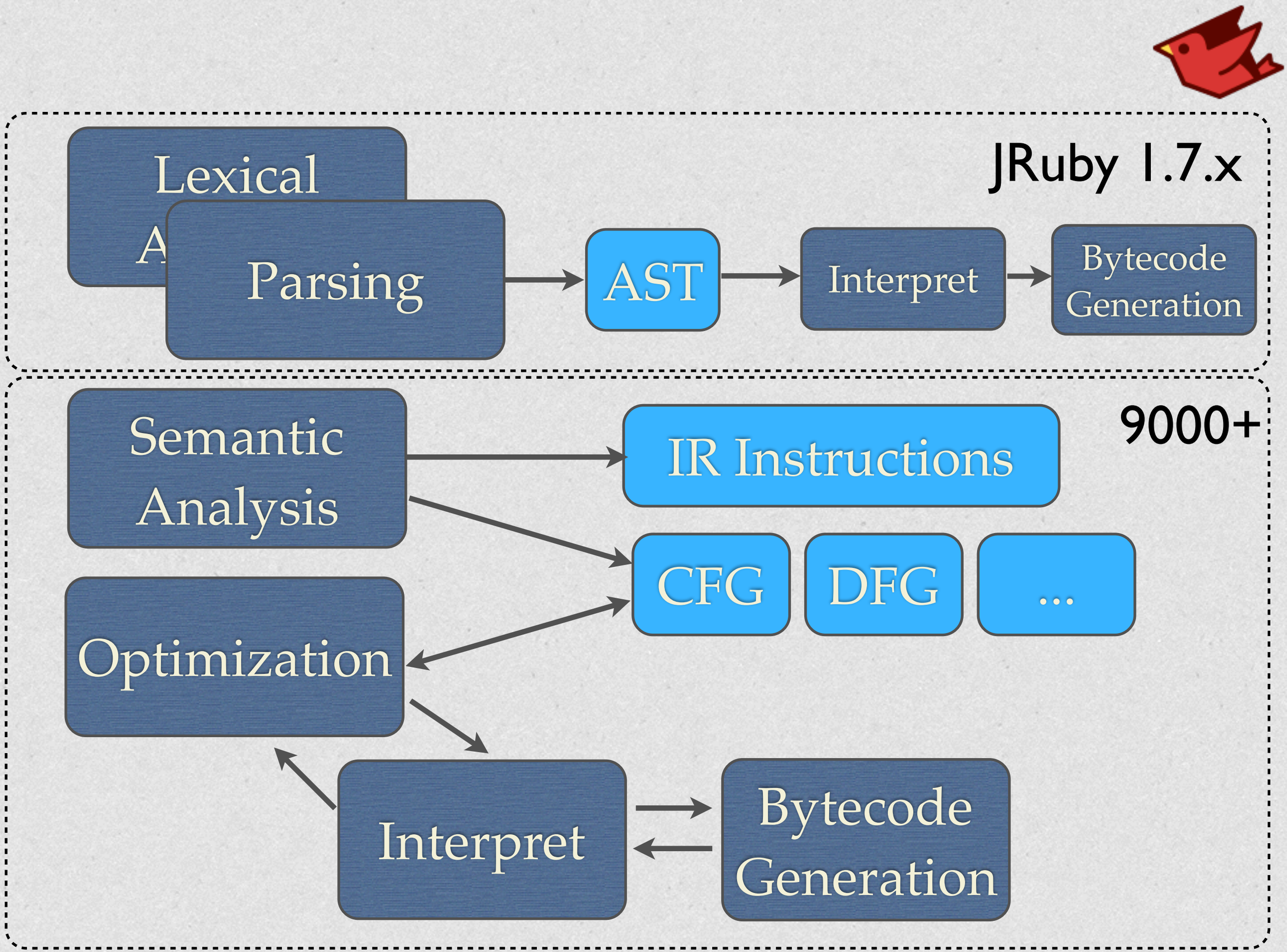
- "Everything's an object"
- Tracing and debugging APIs
- Pervasive use of closures
- Mutable literal strings

JRuby 9000

- Mixed mode runtime (now with tiers!)
- Lazy JIT to JVM bytecode
- byte[] strings and regular expressions
- Lots of native integration via FFI
- 9.0.5.0 is current

New IR

- Optimizable intermediate representation
- AST to semantic IR
- Traditional compiler design
- Register machine
- SSA-ish where it's useful


JRuby 1.7.x
Lexical
A
Parsing
AST
Interpret
Bytecode
Generation
9000+
Semantic
Analysis
IR Instructions
CFG
DFG
...
Optimization
Interpret
Bytecode
Generation

Semantic Analysis → IR Instructions

Register-based

```
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

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

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

→

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

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

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

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

→

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

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

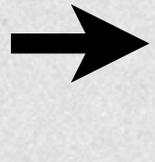
→

```
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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

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



```
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)
```

Optimization

-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)
```

Optimization

-Xir.passes=LocalOptimizationPass,
DeadCodeElimination

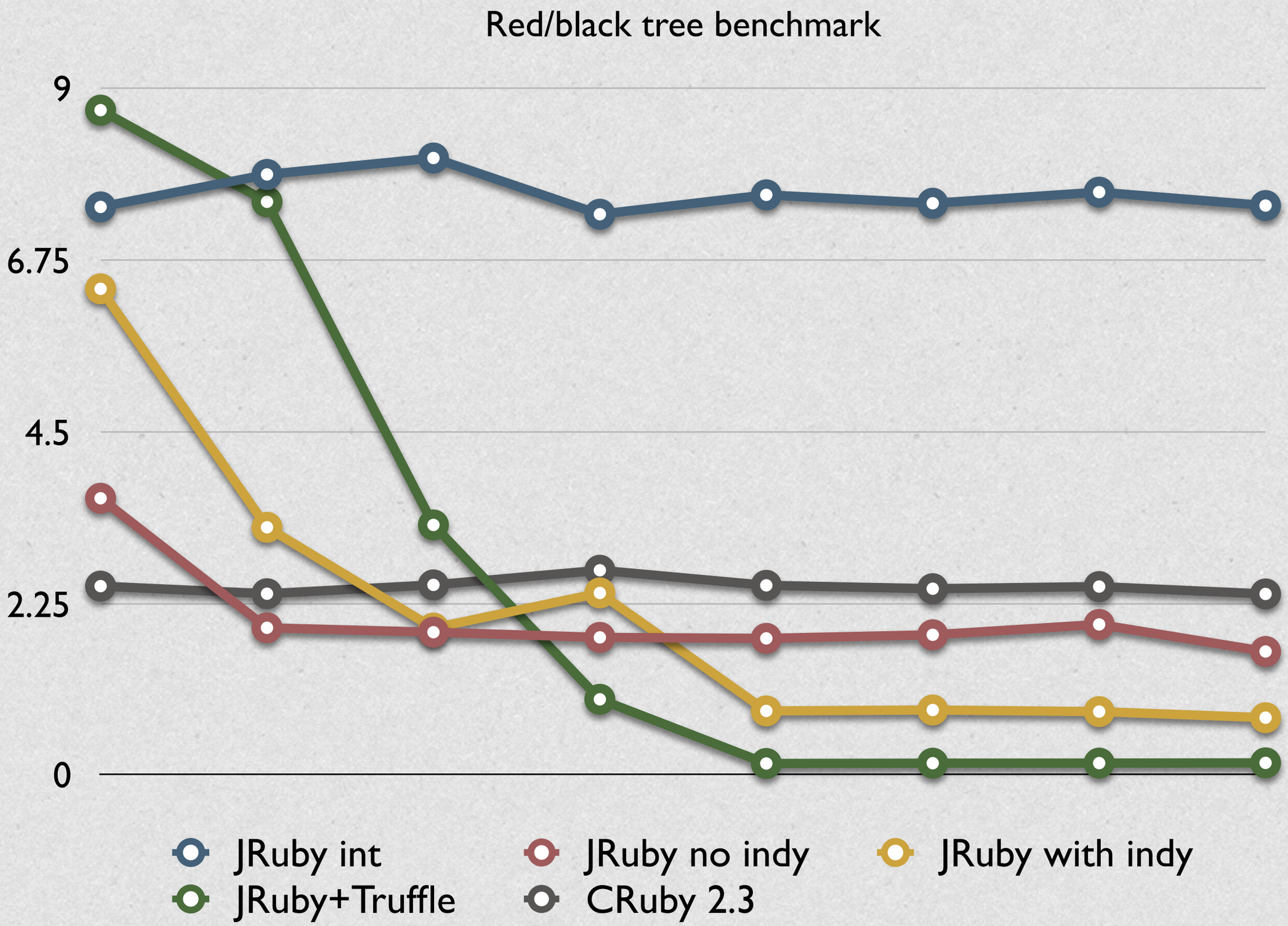
```
0  check_arity(2, 0, -1)
1  a = recv_pre_reqd_arg(0)
4  thread_poll
7  line_num(2)
8  %v_0 = call(:+, a, [1])
9  d = copy(%v_0)
10 return(%v_0)
```

Tiers in the Rain

- Tier 1: Simple interpreter (no passes run)
- Tier 2: Full interpreter (static optimization)
- Tier 3: Full interpreter (profiled optz)
- Tier 4: JVM bytecode (static)
- Tier 5: JVM bytecode (profiled)
- Tiers 6+: Whatever JVM does from there

Truffle?

- Write your AST + specializations
- AST rewrites as it runs
- Eventually emits Graal IR (i.e. not JVM)
- Very fast peak perf on benchmarks
- Poor startup, warmup, memory use
- Year(s) left until generally usable


Red/black tree benchmark
9
6.75
4.5
2.25
0
JRuby int
JRuby+Truffle
JRuby no indy
CRuby 2.3
JRuby with indy

Why Not Just JVM?

- JVM is great, but missing many things
- I'll mention some along the way

Current Optimizations

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

define_method

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

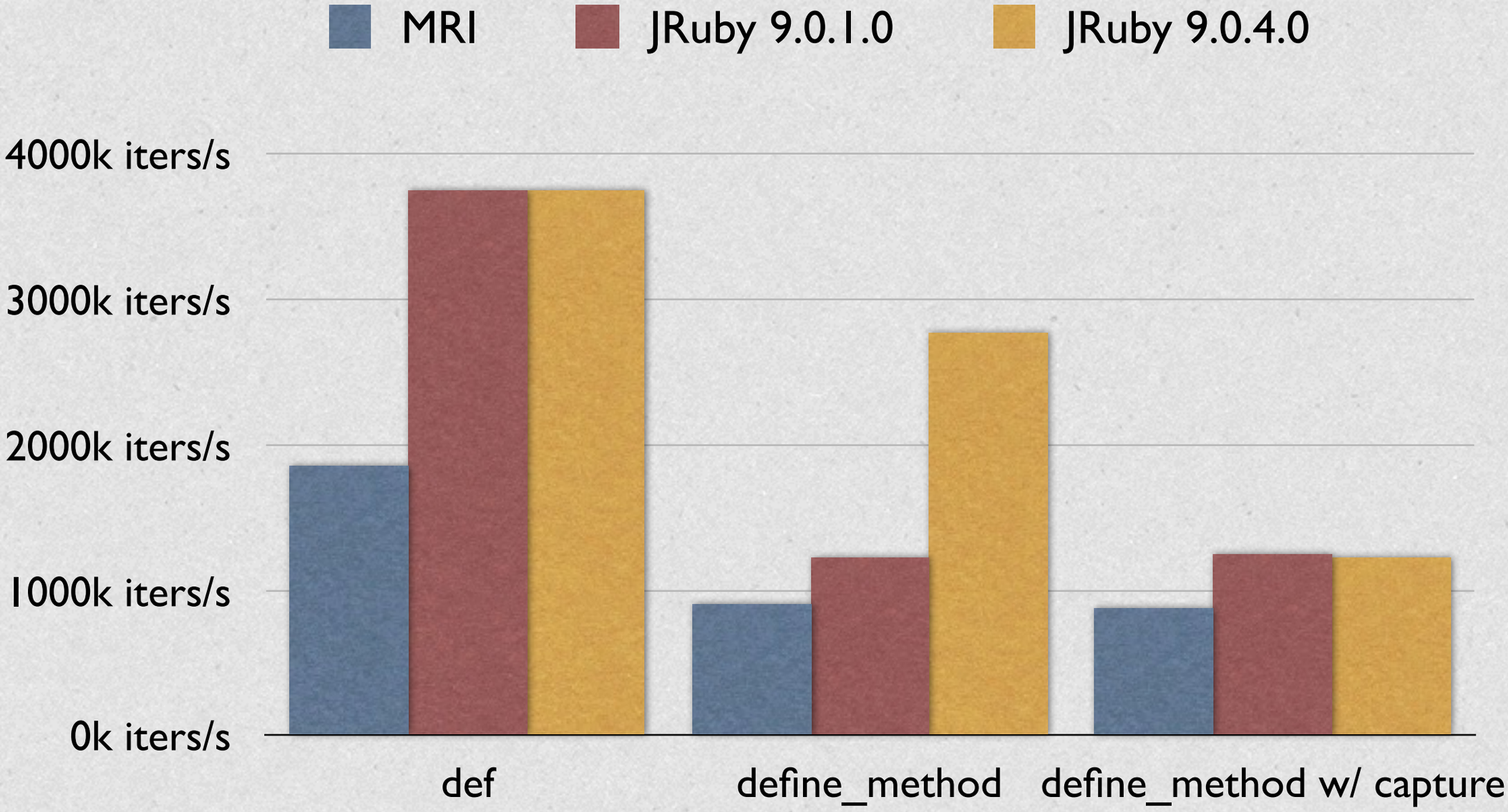
```ruby
names.each do |name|
  define_method(name) { send :"do_#{name}" }
end
```

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!



JVM?

- Missing feature: access to call frames
- No way to expose local variables
- Therefore, have to use heap
- Allocation, loss of locality

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

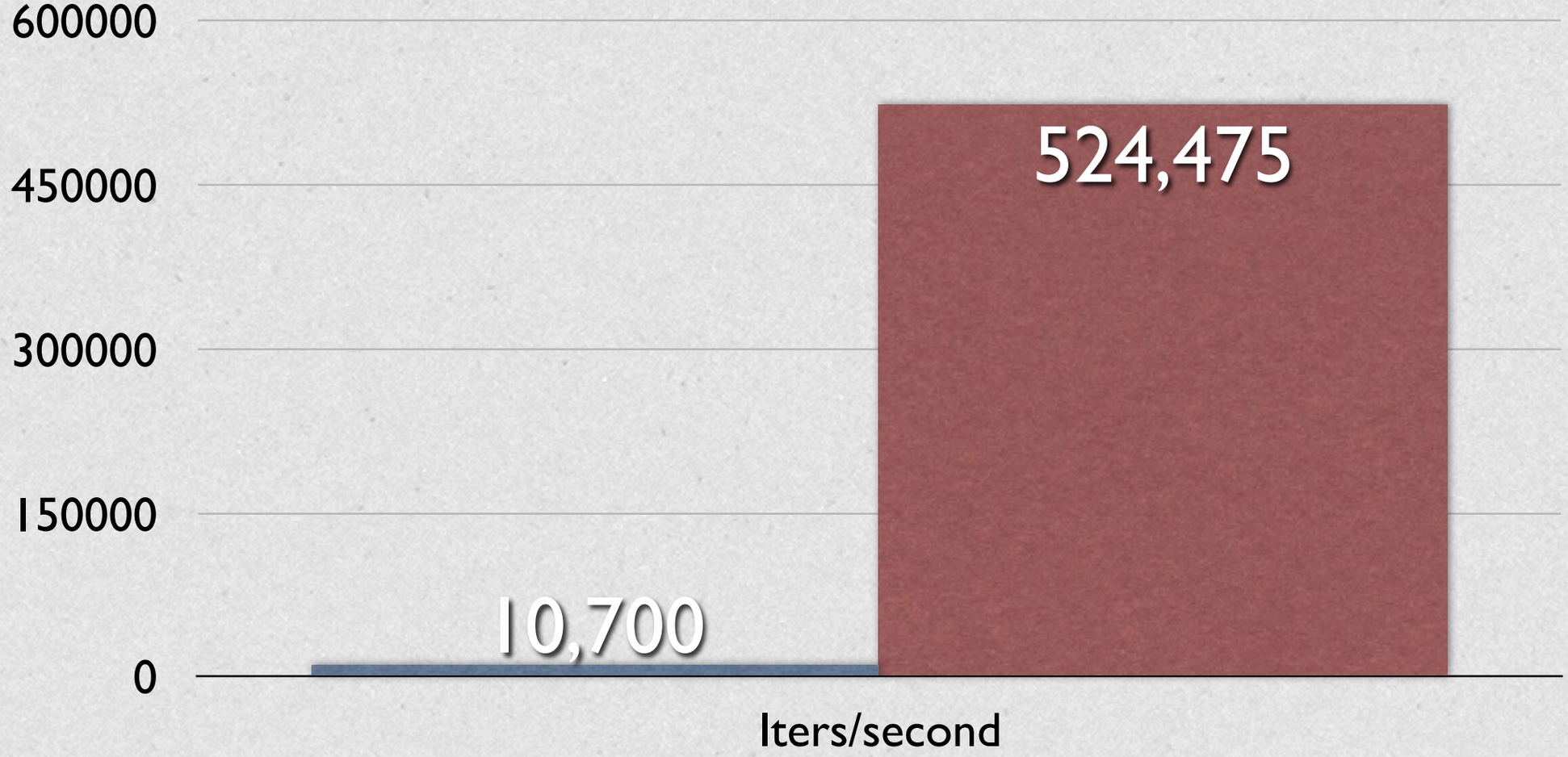
```
Converters  = { integer:    lambda { |f|
                    Integer(f) rescue f
                },
                float:      lambda { |f|
                    Float(f) rescue f
                },
                ...
```

All trivial rescues, no traces needed.

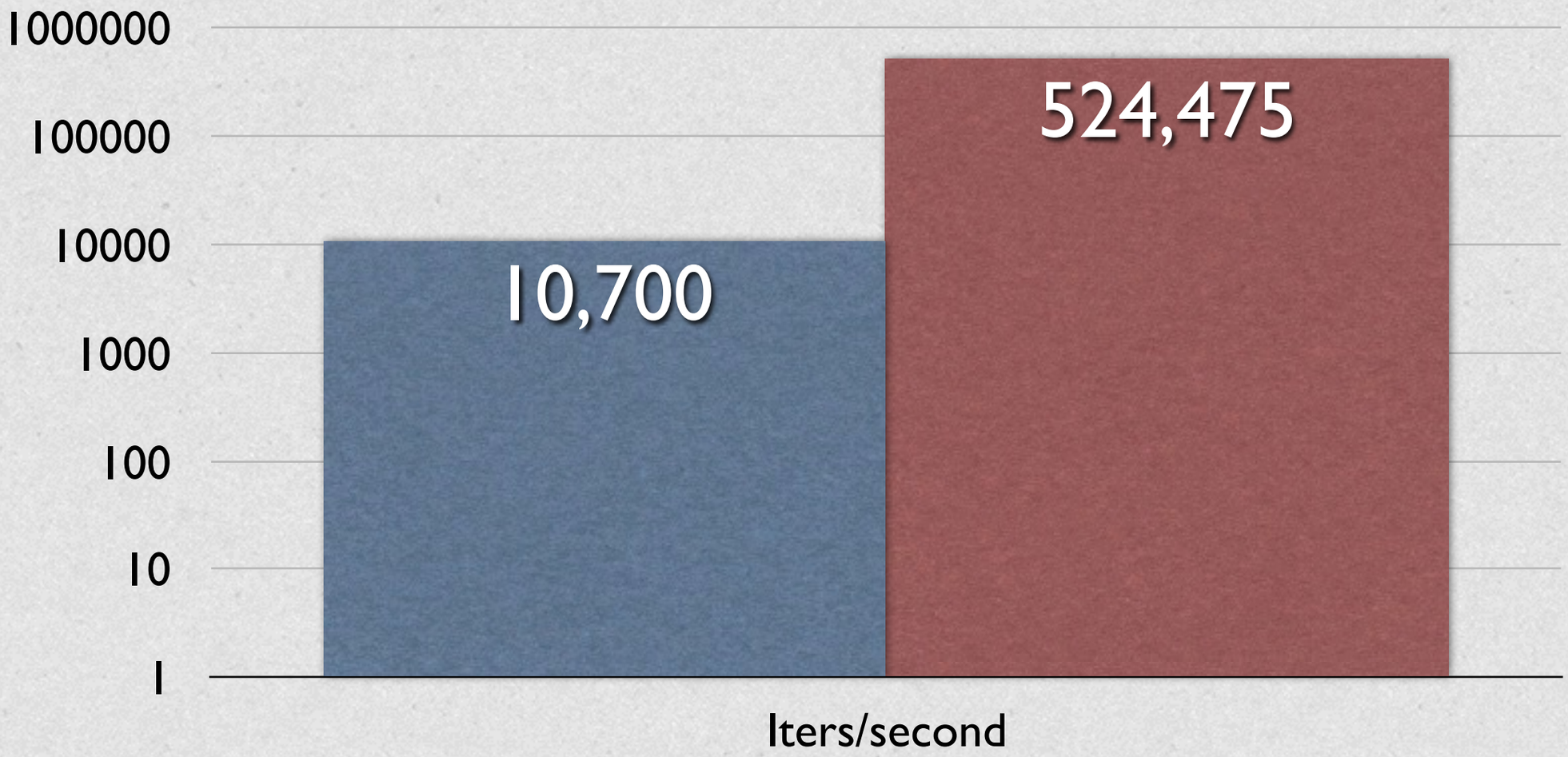
Strategy

- Inspect rescue block
- If simple expression...
  - Thread-local requiresBacktrace = false
  - Backtrace generation short circuited
- Reset to true on exit or nontrivial rescue

Simple rescue Improvement



Much Better!


1000000
100000
10000
1000
100
10
1
10,700
524,475
Iters/second

JVM?

- Horrific cost for stack traces
- Only eliminated if inlined
- Disabling is not really an option

Work In Progress

Object Shaping

- Ruby instance vars allocated dynamically
- JRuby currently grows an array
- We have code to specialize as fields
  - Working, tested
  - Probably next release

```
public class RubyObjectVar2 extends ReifiedRubyObject {
    private Object var0;
    private Object var1;
    private Object var2;

    public RubyObjectVar2(Ruby runtime, RubyClass metaClass) {
        super(runtime, metaClass);
    }

    @Override
    public Object getVariable(int i) {
        switch (i) {
            case 0: return var0;
            case 1: return var1;
            case 2: return var2;
            default: return super.getVariable(i);
        }
    }

    public Object getVariable0() {
        return var0;
    }

...

    public void setVariable0(Object value) {
        ensureInstanceVariablesSettable();
        var0 = value;
    }

...

}
```

JVM?

- No way to truly generify fields
  - Valhalla will be useful here
- No way to grow an object

Inlining

- 900 pound gorilla of optimization
  - shove method/closure back to callsite
    - specialize closure-receiving methods
    - eliminate call protocol
- We know Ruby better than the JVM

JVM?

- JVM will inline for us, but...
  - only if we use invokedynamic
  - and the code isn't too big
  - and it's not polymorphic
  - and we're not a closure (lambdas too!)
  - and it feels like it

Today's Inliner

```ruby
def decrement_one(i)
  i - 1
end

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

```ruby
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

```ruby
def decrement_one(i)
  i - 1
end

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

```ruby
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
```

Today's Inliner

```ruby
def decrement_one(i)
  i - 1
end

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

```ruby
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
```

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

Inlining a Closure

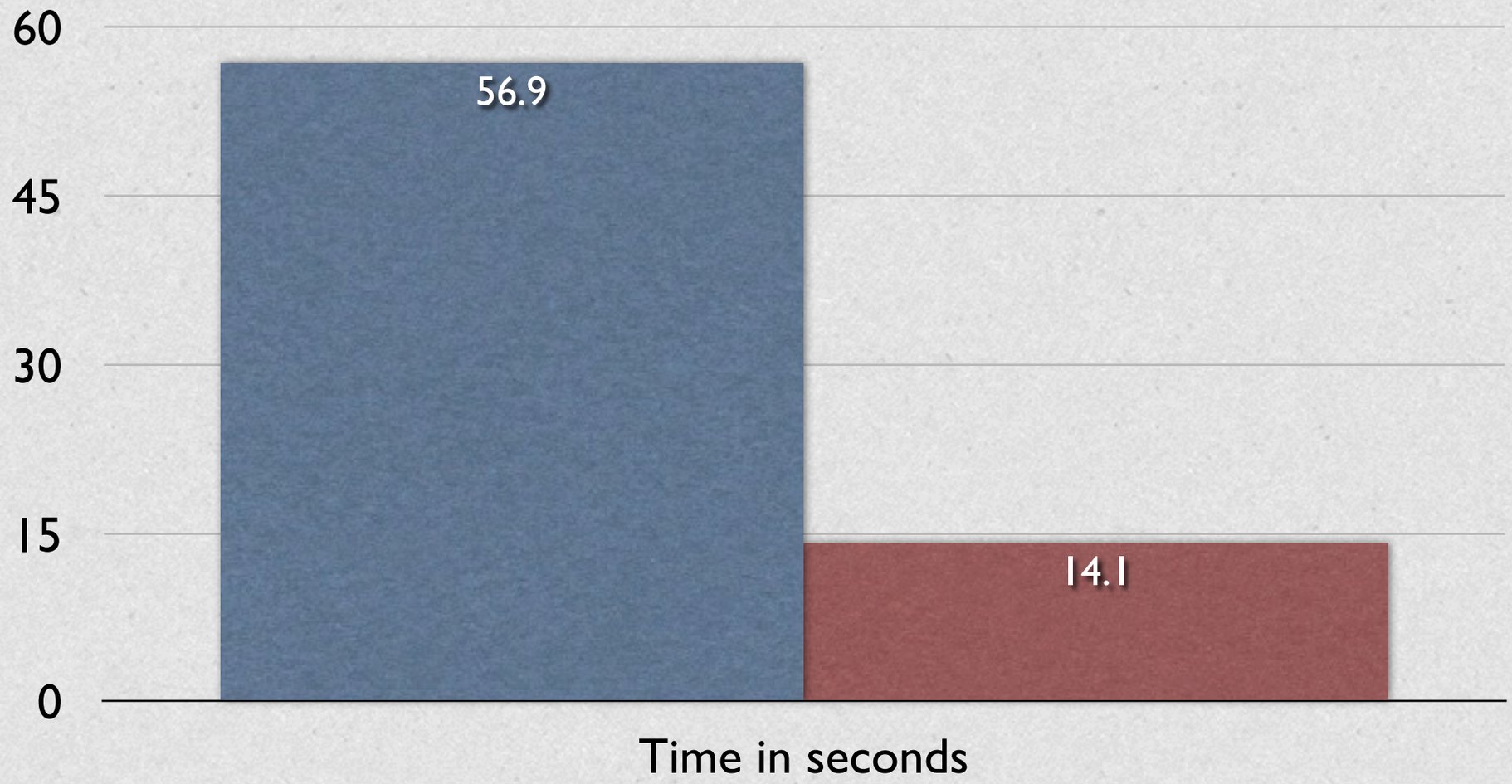
```
def small_loop(i)
  k = 10
  while k > 0
    k = yield(k)
  end
  i - 1
end

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!



Profiling

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

* Fragile and buggy!

Interpreter FTW!

- Deopt is much simpler with interpreter
  - Collect local vars, instruction index
  - Raise exception to interpreter, keep going
- Much cheaper than resuming bytecode

Numeric Specialization

- "Unboxing"
- Ruby: everything's an object
  - Tagged pointer for Fixnum, Float
- JVM: references OR primitives
- Need to optimize numerics as primitive

JVM?

- Escape analysis is inadequate (today)
- Hotspot will eliminate boxes if...
  - All code inlines
  - No (unfollowed?) branches in the code
- Dynamic calls have type guards
- Fixnum + Fixnum has overflow check

```ruby
def looper(n)
  i = 0
  while i < n
    do_something(i)
    i += 1
  end
end
```

Specialize n, i to long

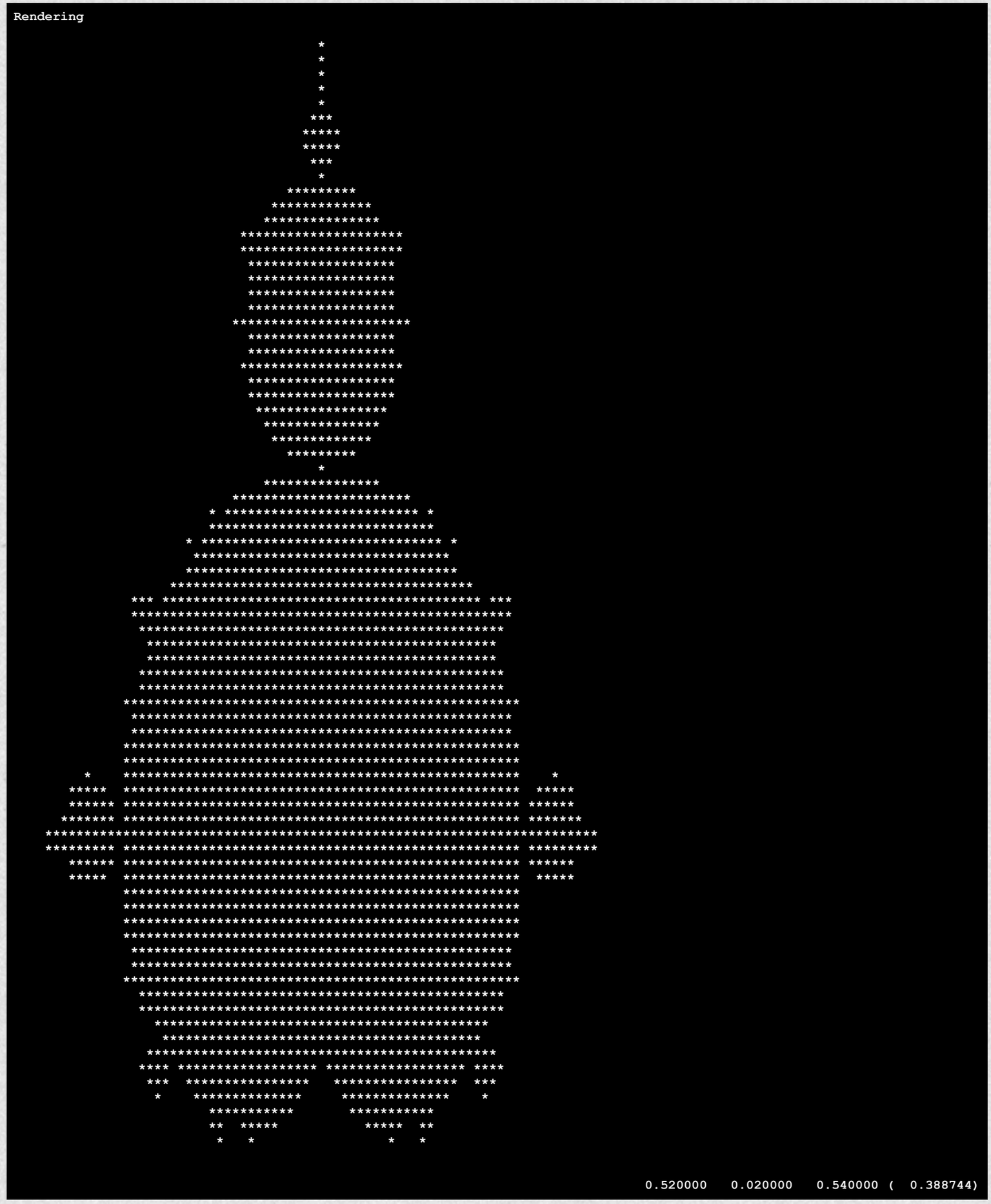
```ruby
def looper(long n)
  long i = 0
  while i < n
    do_something(i)
    i += 1
  end
end
```

```ruby
def looper(n)
  i = 0
  while i < n
    do_something(i)
    i += 1
  end
end
```

Deopt to object version if n or i + 1 is not Fixnum

Unboxing Today

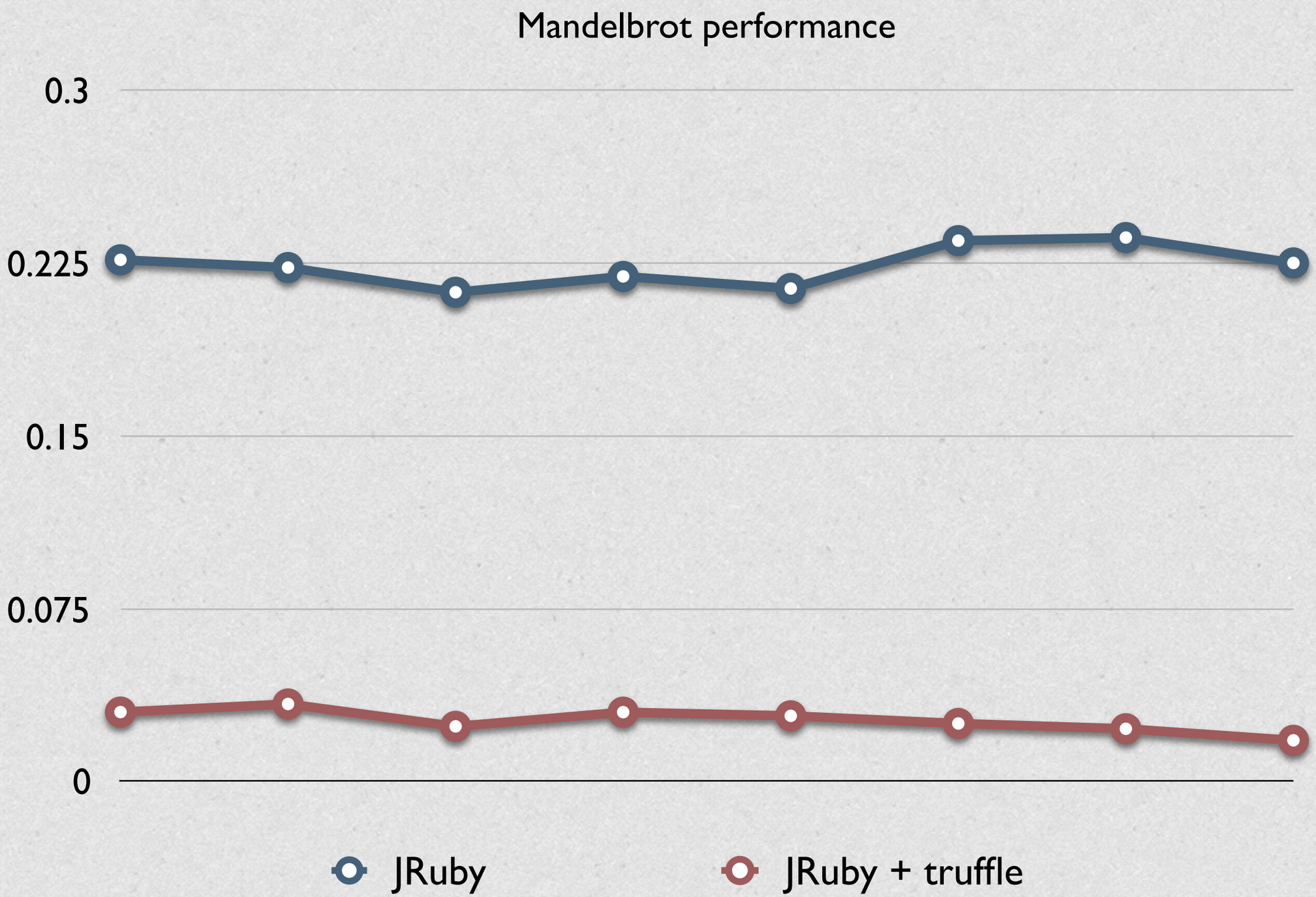
- Working prototype
- No deopt
- No type guards
- No overflow check for Fixnum/Bignum

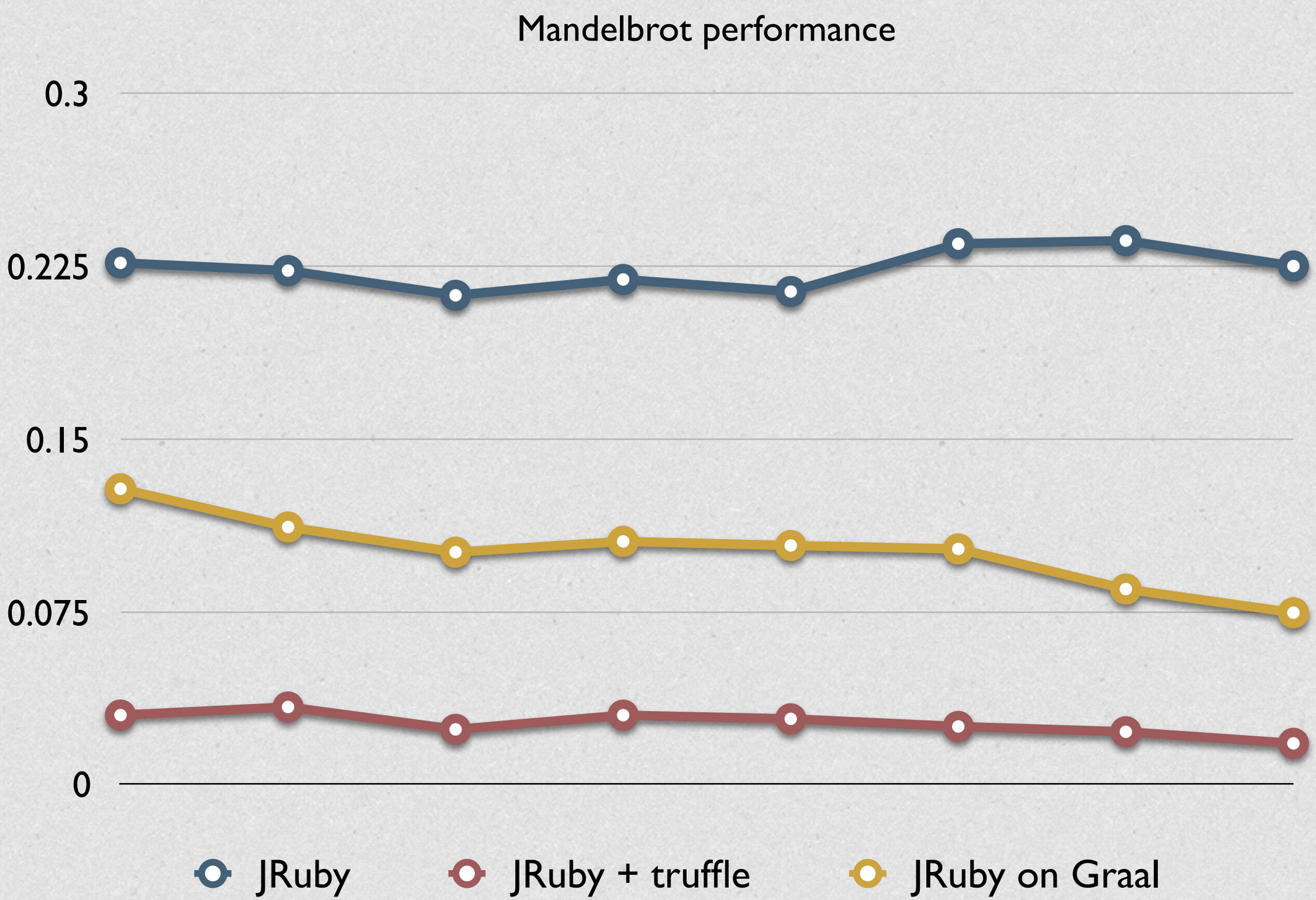

Rendering
0.520000 0.020000 0.540000 (0.388744)

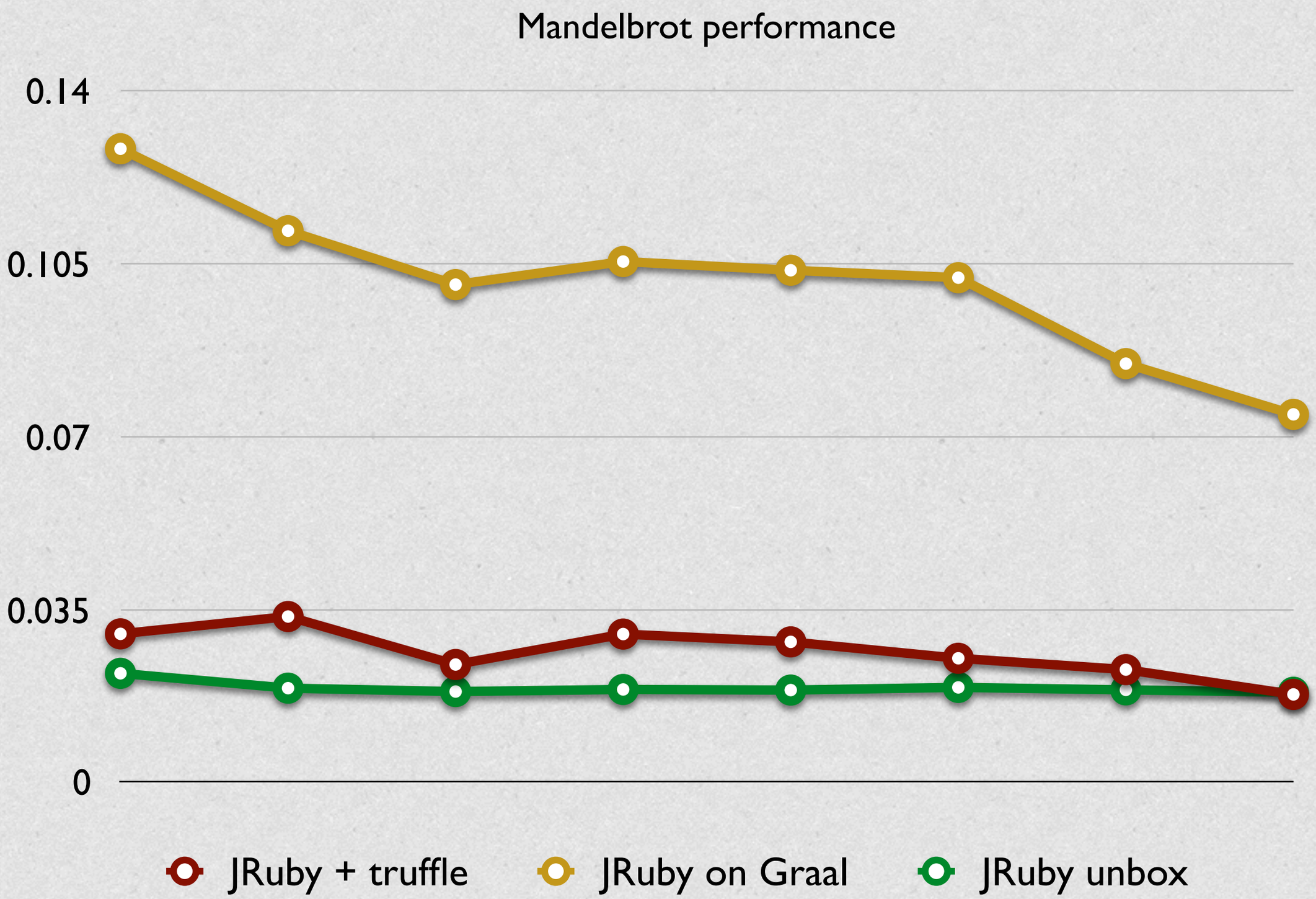
```
def iterate(x,y)
  cr = y-0.5
  ci = x
  zi = 0.0
  zr = 0.0
  i = 0
  bailout = 16.0
  max_iterations = 1000

  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
```

```
basic_block #3: _LOOP_BEGIN_0:24
  00:                          line_num(lineNumber: 37)
  01:                          thread_poll(onBackEdge: true)
  02:                %v_5 := call_1f(%t_i_24, fix<1>, fixNum: 1, callType: NORMAL, name: +, potentiallyRefined: false)
  03:              %t_i_24 := copy(%v_5)
  04:                          line_num(lineNumber: 38)
  05:                %v_6 := call_1o(%t_zr_23, %t_zi_22, callType: NORMAL, name: *, potentiallyRefined: false)
  06:           %t_temp_27 := copy(%v_6)
  07:                          line_num(lineNumber: 39)
  08:                %v_7 := call_1o(%t_zr_23, %t_zr_23, callType: NORMAL, name: *, potentiallyRefined: false)
  09:            %t_zr2_28 := copy(%v_7)
  10:                          line_num(lineNumber: 40)
  11:                %v_8 := call_1o(%t_zi_22, %t_zi_22, callType: NORMAL, name: *, potentiallyRefined: false)
  12:            %t_zi2_29 := copy(%v_8)
  13:                          line_num(lineNumber: 41)
  14:                %v_9 := call_1o(%t_zr2_28, %v_8, callType: NORMAL, name: -, potentiallyRefined: false)
  15:               %v_10 := call_1o(%v_9, %t_cr_20, callType: NORMAL, name: +, potentiallyRefined: false)
  16:            %t_zr_23 := copy(%v_10)
  17:                          line_num(lineNumber: 42)
  18:               %v_11 := call_1o(%t_temp_27, %t_temp_27, callType: NORMAL, name: +, potentiallyRefined: false)
  19:               %v_12 := call_1o(%v_11, %t_ci_21, callType: NORMAL, name: +, potentiallyRefined: false)
  20:            %t_zi_22 := copy(%v_12)
  21:                          line_num(lineNumber: 43)
  22:               %v_13 := call_1o(%t_zi2_29, %t_zr2_28, callType: NORMAL, name: +, potentiallyRefined: false)
  23:               %v_14 := call_1o(%v_13, %t_bailout_25, callType: NORMAL, name: >, potentiallyRefined: false)
  24:                          b_false(ipc<LBL_19:53>, %v_14, jumpTarget: LBL_19:53, value: %v_14)

basic_block #5: LBL_31:-1
  00:                          return(%t_i_24)
```

```
basic_block #3: _LOOP_BEGIN_0:24
  00:                          line_num(lineNumber: 37)
  01:                          thread_poll(onBackEdge: true)
  02:                 %i_3 := iadd(%i_1, rawfix<1>)
  03:                 %i_1 := copy(%i_3)
  04:                          line_num(lineNumber: 38)
  05:                 %f_5 := fmul(%f_1, %f_0)
  06:                 %f_6 := copy(%f_5)
  07:                          line_num(lineNumber: 39)
  08:                 %f_7 := fmul(%f_1, %f_1)
  09:                 %f_8 := copy(%f_7)
  10:                          line_num(lineNumber: 40)
  11:                 %f_9 := fmul(%f_0, %f_0)
  12:                %f_10 := copy(%f_9)
  13:                          line_num(lineNumber: 41)
  14:                %f_11 := fsub(%f_8, %f_9)
  15:                %f_12 := fadd(%f_11, %f_4)
  16:                 %f_1 := copy(%f_12)
  17:                          line_num(lineNumber: 42)
  18:                %f_13 := fadd(%f_6, %f_6)
  19:                %f_14 := fadd(%f_13, %f_3)
  20:                 %f_0 := copy(%f_14)
  21:                          line_num(lineNumber: 43)
  22:                %f_15 := fadd(%f_10, %f_8)
  23:                 %b_1 := fgt(%f_15, %f_2)
  24:                          b_false(ipc<LBL_19:53>, %b_1, jumpTarget: LBL_19:53, value: %b_1)

basic_block #5: LBL_31:-1
  00:               %t_i_24 := box_fixnum(%i_1)
  01:                          return(%t_i_24)
```


Mandelbrot performance
0.3
0.225
0.15
0.075
0
JRuby
JRuby + truffle


Mandelbrot performance
0.3
0.225
0.15
0.075
0
JRuby
JRuby + truffle
JRuby on Graal


Mandelbrot performance
0.14
0.105
0.07
0.035
0
JRuby + truffle
JRuby on Graal
JRuby unbox

When?

- Object shape should be in 9.1
- Profiling, inlining mostly need testing
- Specialization needs guards, deopt
- Active work over next 6-12mo

Summary

- JVM is great, but we need more
  - Partial EA, frame access, specialization
  - Gotta stay ahead of these youngsters!
- JRuby 9000 is a VM on top of a VM
- We believe we can match Truffle
  - (for a large range of optimizations)

Thank You

- Charles Oliver Nutter
- @headius
- headius@headius.com