Accelerating Ruby with LLVM

Evan Phoenix

Oct 2, 2009

RUBY

RUBY

Strongly, dynamically typed

RUBY

Unified Model

RUBY

Everything is an object

RUBY

```
3.class # => Fixnum
```

RUBY

Every code context is equal

RUBY

Every context is a method

RUBY

Garbage Collected

RUBY

A lot of syntax

RUBY

Strongly, dynamically typed
Unified model
Everything is an object
3.class

Every code context is equal
Every context is a method
Garbage collected
A lot of syntax

Rubinius

Rubinius

Started in 2006

Rubinius

Build a ruby environment for fun

Rubinius

Unlike most “scripting” languages,
write as much in ruby as possible

Rubinius

Core functionality of perl/python/ruby in C,
NOT in their respective language.

Rubinius

`C => ruby => C => ruby`

Rubinius

Language boundaries suck

Rubinius

Started in 2006

Built for fun

Turtles all the way down

Evolution

Evolution

100% ruby prototype running on 1.8

Evolution

Hand translated VM to C

Evolution

Rewrote VM in C++

Evolution

Switch away from stackless

Evolution

Experimented with handwritten assembler for x86

Evolution

Switch to LLVM for JIT

Evolution

100% ruby prototype
Hand translated VM to C
Rewrote VM in C++
Switch away from stackless
Experiment with assembler
Switch to LLVM for JIT

Features

Features

Bytecode VM

Features

Simple interface to native code

Features

Accurate, generational garbage collector

Features

Integrated FFI API

Features

Bytecode VM

Generational GC

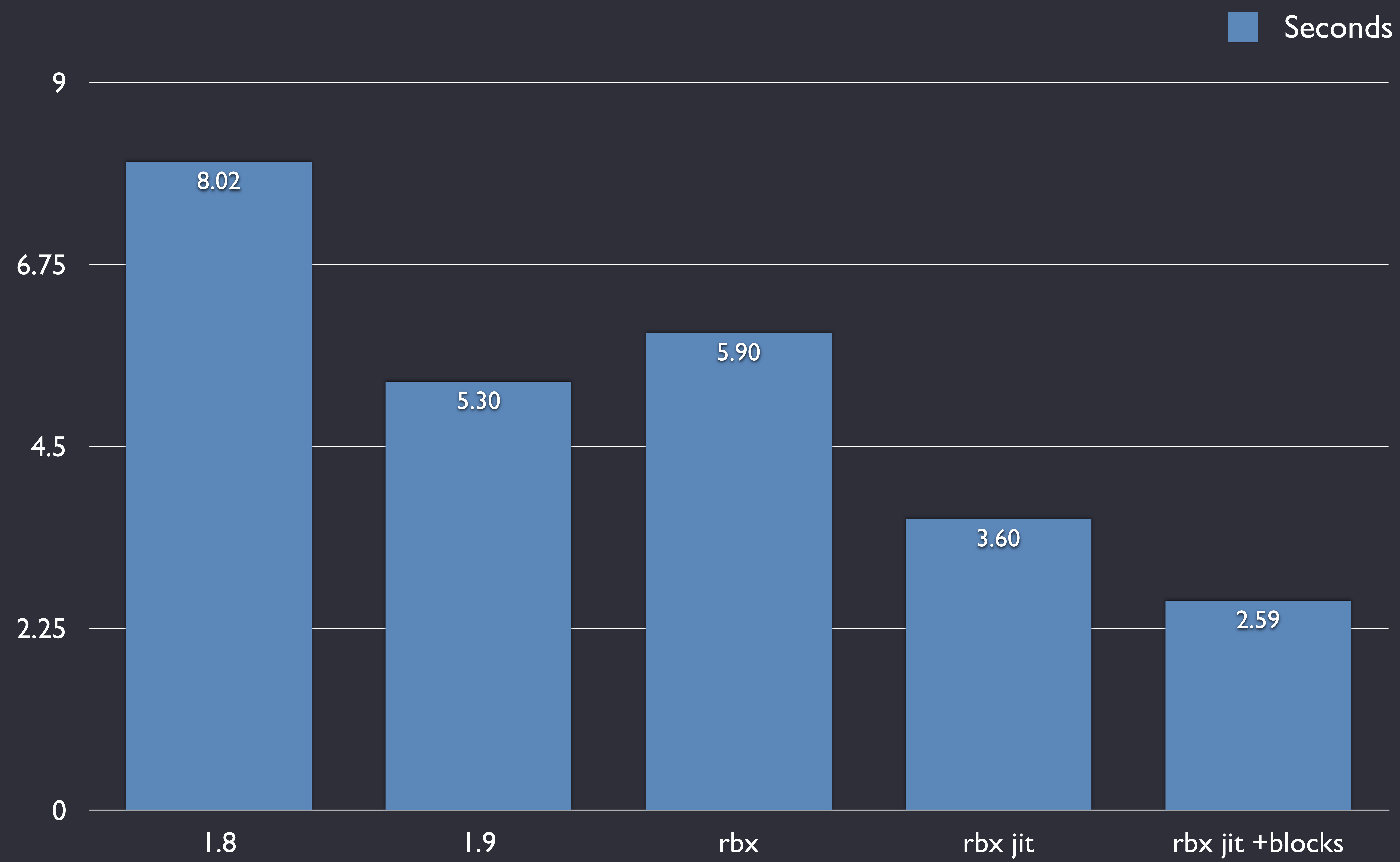
Interface to native code

Integrated FFI

Benchmarks

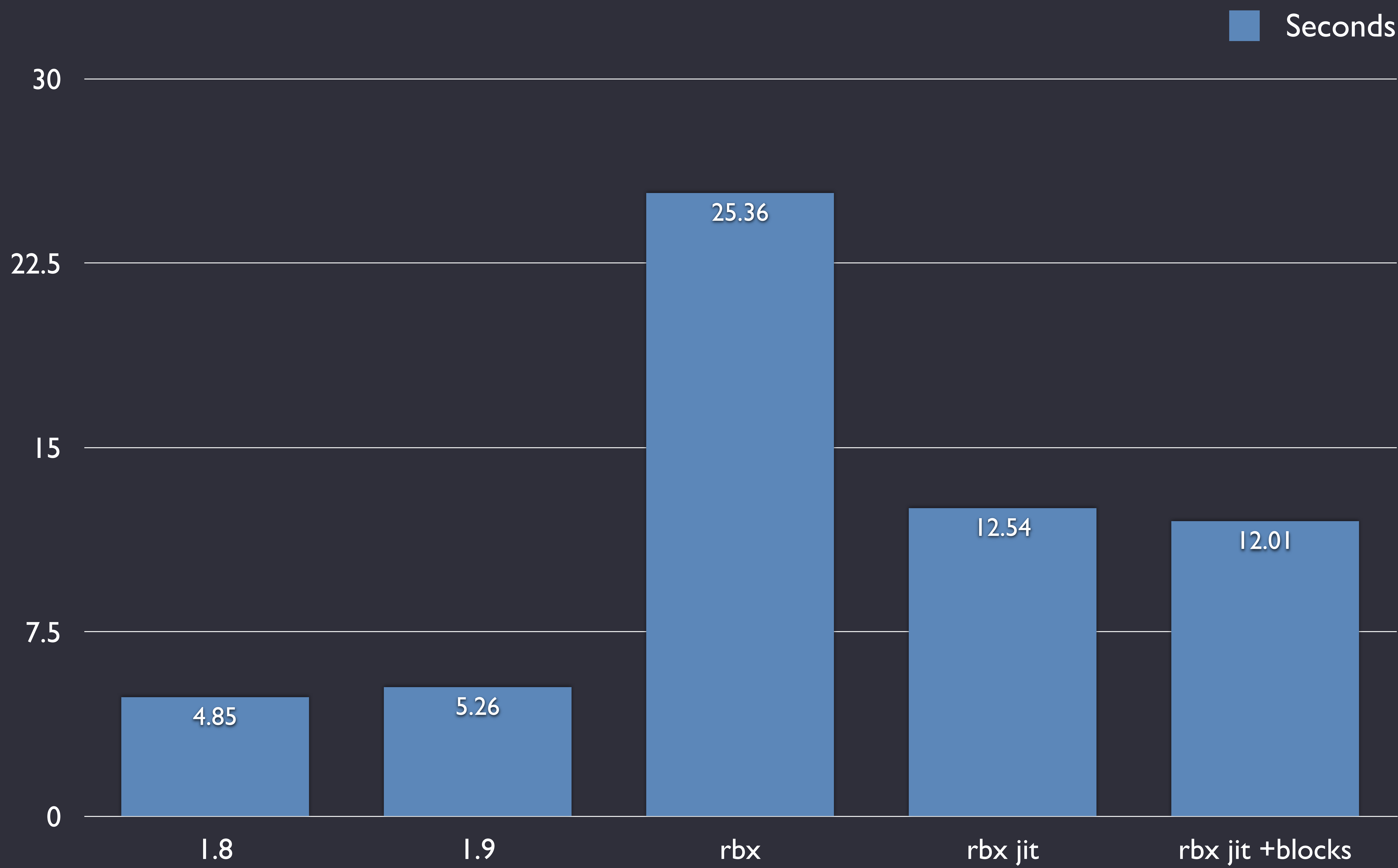
```ruby
def foo()
  ary = []
  100.times { |i| ary << i }
end
```

300,000 times


Seconds
9
6.75
4.5
2.25
0
8.02
5.30
5.90
3.60
2.59
1.8
1.9
rbx
rbx jit
rbx jit +blocks

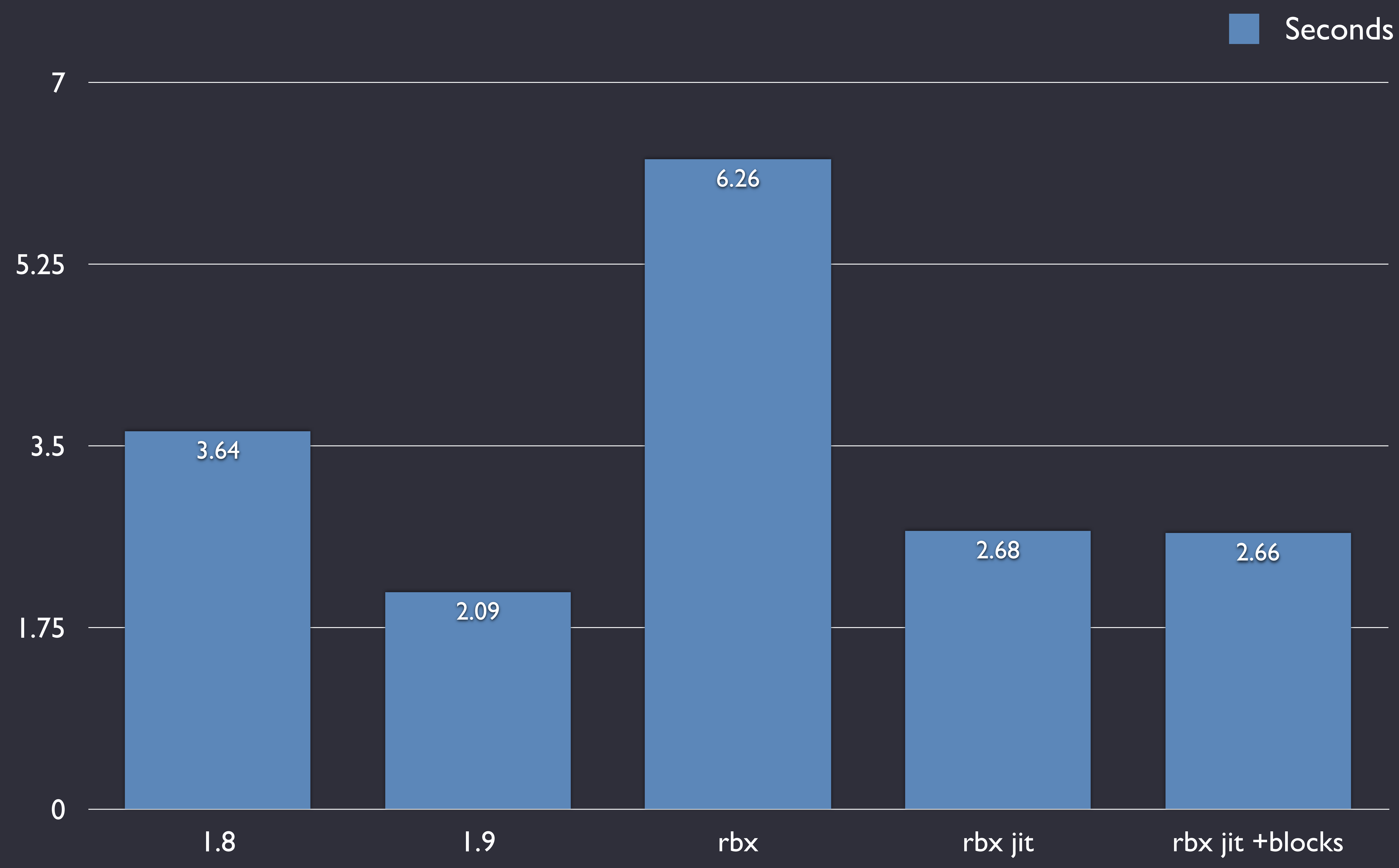
```ruby
def foo()
  hsh = {}
  100.times { |i| hsh[i] = 0 }
end
```

100,000 times


Seconds
30
22.5
15
7.5
0
4.85
5.26
25.36
12.54
12.01
1.8
1.9
rbx
rbx jit
rbx jit +blocks

```
def foo()
  hsh = { 47 => true }
  100.times { |i| hsh[i] }
end
```

100,000 times


Seconds
7
5.25
3.5
1.75
0
3.64
2.09
6.26
2.68
2.66
1.8
1.9
rbx
rbx jit
rbx jit +blocks

Early LLVM Usage

Early LLVM Usage

Compiled all methods up front

Early LLVM Usage

Simple opcode-to-function translation
with inlining

Early LLVM Usage

Startup went from 0.3s to 80s

Early LLVM Usage

Compiled all methods upfront

Simple opcode-to-function translation

Startup from 0.3s to 80s

True JIT

True JIT

JIT Goals

True JIT

JIT Goals

Choose methods that benefit the most

True JIT

JIT Goals

Compiling has minimum impact on performance

True JIT

JIT Goals

Ability to make intelligent frontend decisions

Choosing Methods

Choosing Methods

Simple call counters

Choosing Methods

When counter trips, the fun starts

Choosing Methods

Room for improvement

Choosing Methods

Room for improvement

Increment counters in loops

Choosing Methods

Room for improvement

Weigh different invocations differently

Choosing Methods

Simple counters
Trip the counters, do it

Room for improvement
Increment in loops
Weigh invocations

Which Method?

Which Method?

Leaf methods trip quickly

Which Methods?

Leaf methods trip quickly

Consider the whole callstack

Which Methods?

Leaf methods trip quickly

Pick a parent expecting inlining

Which Method?

Leaf methods trip

Consider the callstack

Find a parent

Minimal Impact

Minimal Impact

After the counters trip

Minimal Impact

Queue the method

Minimal Impact

Background thread drains queue

Minimal Impact

Frontend, passes, codegen in background

Minimal Impact

Install JIT'd function

Minimal Impact

Install JIT'd function

Requires GC interaction

Minimal Impact

Trip the counters	Compile in background
Queue the method	Install function pointer

Good Decisions

Good Decisions

Naive translation yields fixed improvement

Good Decisions

Performance shifts to method dispatch

Good Decisions

Improve optimization horizon

Good Decisions

Inline using type feedback

Good Decisions

Naive translation sucks
Inline using type feedback

Performance in dispatch
Improve optimizations

Type Feedback

Type Feedback

Frontend translates to IR

Type Feedback

Read InlineCache information

Type Feedback

InlineCaches contain profiling info

Type Feedback

Use profiling to drive inlining!

Type Feedback

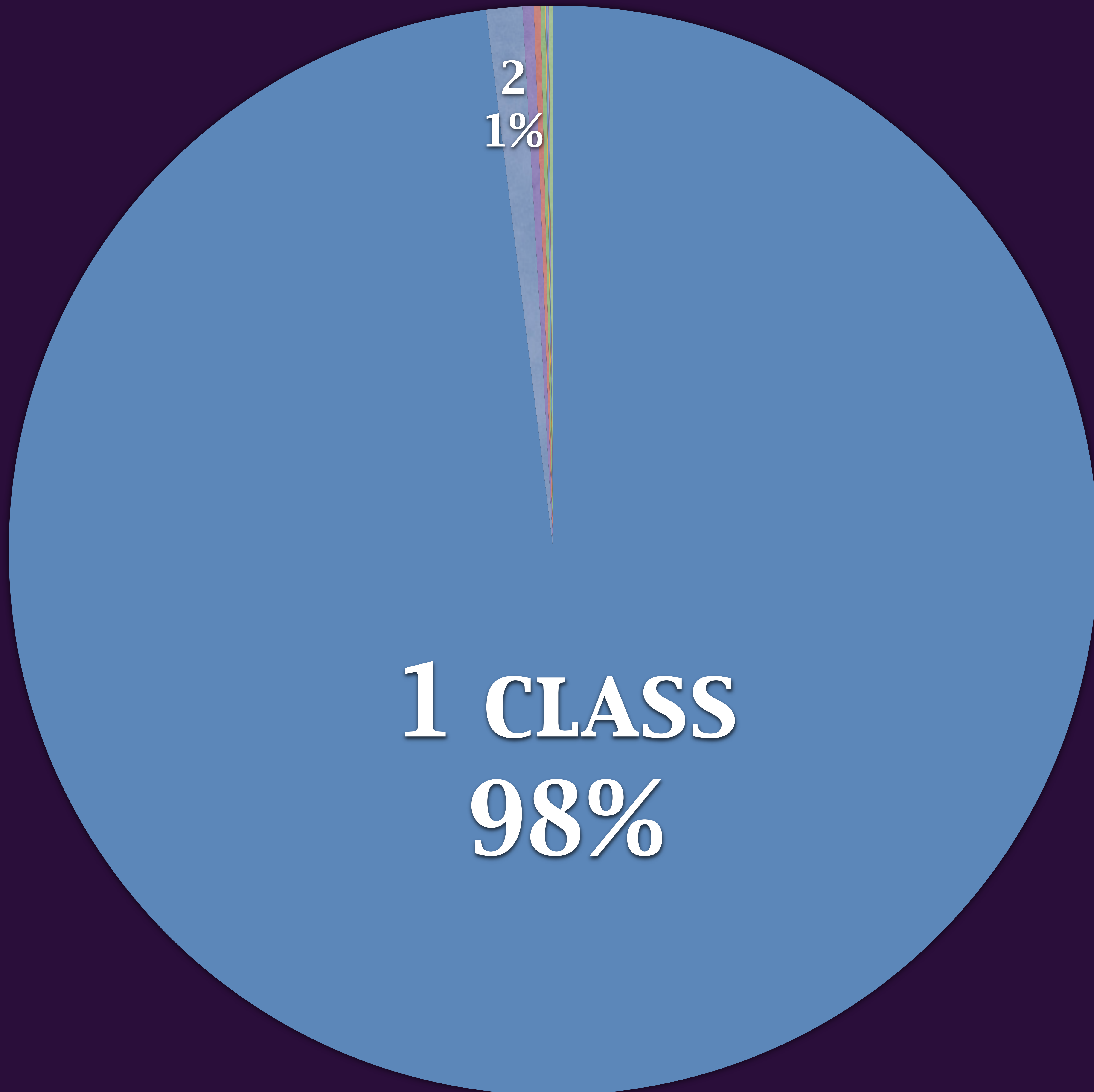
Frontend generates IR
Reads InlineCaches

InlineCaches have profiling
Use profiling to drive inlining!

Inlining

Inlining

Profiling info shows a dominant class


2
1%
1 CLASS
98%

Inlining

Lookup method in compiler

Inlining

For native functions, emit direct call

Inlining

For FFI, inline conversions and call

Inlining

Find dominant class
Lookup method

Emit direct calls if possible

Inlining Ruby

Inlining Ruby

Policy decides on inlining

Inlining Ruby

Drive sub-frontend at call site

Inlining Ruby

All inlining occurs in the frontend

Inlining Ruby

Generated IR preserves runtime data

Inlining Ruby

Generated IR preserves runtime data

GC roots, backtraces, etc

Inlining Ruby

No AST between bytecode and IR

Inlining Ruby

No AST between bytecode and IR

Fast, but limits the ability to generate better IR

Inlining Ruby

Policy decides
Drive sub-frontend

Preserve runtime data
Generates fast, ugly IR

LLVM

LLVM

IR uses operand stack

LLVM

IR uses operand stack

Highlevel data flow not in SSA

LLVM

IR uses operand stack

Passes eliminate redundencies

LLVM

IR uses operand stack

Makes GC stack marking easy

LLVM

IR uses operand stack

nocapture improves propagation

LLVM

Exceptions via sentinal value

LLVM

Exceptions via sentinal value

Nested handlers use branches for control

LLVM

Exceptions via sentinal value

Inlining exposes redundant checks

LLVM

Inline guards

LLVM

Inline guards

Simple type guards

```
if(obj->class->class_id ==
  <integer constant>) {
```

LLVM

Inline guards

Custom AA pass for guard elimination

LLVM

Inline guards

Teach pointsToConstantMemory about...

```
if(obj->class->class_id ==
   <integer constant>) {
```

```
if(obj->class->class_id ==
   <integer constant>) {
```

LLVM

Maximizing constant propagation

LLVM

Maximizing constant propagation

Type failures shouldn't contribute values

```
if(obj->class->class_id == 0x33) {
  val = 0x7;
} else {
  val = send_msg(state, obj, ...);
}
```

```
if(obj->class->class_id == 0x33) {
  val = 0x7;
} else {
  return uncommon(state);
}
```

LLVM

Maximizing constant propagation

Makes JIT similar to tracing

LLVM

Use overflow intrinsics

LLVM

Use overflow intrinsics

Custom pass to fold constants arguments

LLVM

AA knowledge for tagged pointers

LLVM

AA knowledge of tagged pointers

0x5 is *2* as a tagged pointer

LLVM

Not in SSA form
Simplistic exceptions
Inlining guards

Maximize constants
Use overflow
Tagged pointer AA

Issues

Issues

How to link with LLVM?

Issues

How to link with LLVM?

An important SCM issue

Issues

Ugly, confusing IR from frontend

Issues

instcombine confuses basicaa

Issues

Operand stack confuses AA

Issues

Inability to communicate semantics

```
Object* new_object(state)
```

Returned pointer aliases nothing

Only modifies state

If return value is unused, remove the call

Semi-pure?

Issues

Ugly IR

Linking with LLVM

AA confusion

Highlevel semantics

Thanks!

http://rubini.us

ephoenix@engineyard.com