
# Ruby on the Modern JVM

# Who Am I

- Charles Oliver Nutter
- @headius(@mastodon.social)
- headius@headius.com
- JRuby developer since 2004
- Full-time JRuby and JVM language advocate since 2006
- Excited to be here complaining about JVM once again

# Q/A

- Probably won't have spare time for interactive Q/A
- Post in FOSDEM Java Matrix room and I will answer
- Or find me other ways


JRuby

# JRuby

- Ruby language for the JVM
  - Next release drops Java 8 for 17 or 21 minimum
- In development since 2001, running Rails since 2006
- Only widely-deployed alternative Ruby
- Most successful off-platform JVM language?

# Riding the Rails

- Almost nothing worked in JRuby ca. 2005
- We started with basic tools and filled in the blanks
- Minimal compliance tests available so we ran everything we could
- And we quickly ran into challenges adapting Ruby to JVM
  - Now the fun starts 😄

# Challenges of Ruby on JVM

- Strings, encodings, regular expressions
- POSIX IO, native library requirements
- Dynamic everything
- Coroutines/microthreads/fibers
- Startup and warmup time

# Challenges help us grow

# Strings and Regex

# Regular Expressions

- Heavily used throughout Ruby
- Java's Regex crashes on some expressions (e.g. /(a|b)*/)
- We tried every Java regex engine we could find
  - JRegex was the best for Java String
  - Java String was eventually insufficient

```
$ jruby -w -e 'java.util.regex.Pattern.matches("(a|b)*", "a" * 10000 + "b")'
Error: Your application used more stack memory than the safety cap of 2048K.
Specify -J-Xss####k to increase it (#### = cap size in KB).
java.lang.StackOverflowError
 at java.util.regex.Pattern$Loop.match(java/util/regex/Pattern.java:5074)
 at java.util.regex.Pattern$GroupTail.match(java/util/regex/Pattern.java:5000)
 at java.util.regex.Pattern$BranchConn.match(java/util/regex/Pattern.java:4878)
 at java.util.regex.Pattern$BmpCharProperty.match(java/util/regex/Pattern.java:4134)
 at java.util.regex.Pattern$Branch.match(java/util/regex/Pattern.java:4914)
 at java.util.regex.Pattern$GroupHead.match(java/util/regex/Pattern.java:4969)
 at java.util.regex.Pattern$Loop.match(java/util/regex/Pattern.java:5078)
 at java.util.regex.Pattern$GroupTail.match(java/util/regex/Pattern.java:5000)
 at java.util.regex.Pattern$BranchConn.match(java/util/regex/Pattern.java:4878)
 at java.util.regex.Pattern$BmpCharProperty.match(java/util/regex/Pattern.java:4134)
 at java.util.regex.Pattern$Branch.match(java/util/regex/Pattern.java:4914)
 at java.util.regex.Pattern$GroupHead.match(java/util/regex/Pattern.java:4969)
 at java.util.regex.Pattern$Loop.match(java/util/regex/Pattern.java:5078)
 at java.util.regex.Pattern$GroupTail.match(java/util/regex/Pattern.java:5000)
 at java.util.regex.Pattern$BranchConn.match(java/util/regex/Pattern.java:4878)
 at java.util.regex.Pattern$BmpCharProperty.match(java/util/regex/Pattern.java:4134)
  ...
```

# Joni

- Java Oniguruma ported from C by Marcin Mielżyński
  - Bytecode-based register machine, no stack issues
  - byte[] matching (also ported to char[] for Nashorn)
  - Pluggable character encodings, grammars
- https://github.com/jruby/joni

# One String to Rule Them All

- Ruby's String is a byte[] plus an Encoding
  - Mutable by default
  - "Binary" is considered an encoding (ASCII-8BIT)
- Emulating this with java.lang.String was impossible
  - Implemented our own String and Encoding logic 🤯
  - Complicates interop with Java

# JCodings

- byte[]-based character encoding and transcoding framework
  - Ported from CRuby's own "multilingualization" backend (M17N)
- All major encodings supported, plus most weird ones
- https://github.com/jruby/jcodings

# JVM Today: Strings and Regex

- Java's String moved from char[] to byte[]
  - Only ISO-8859-1 or UTF-16 supported
  - Lower cost for ASCII strings from Ruby, but UTF-8?
- java.util.Regex still blows stack
  - No plans to replace it?

# Dynamic Language JIT

# JIT Compiler

- Interpretation worked, but was much too slow
  - All state on heap, no inlining of code, JVM could not optimize
- First mixed-mode JIT on JVM, first native JIT for Ruby
  - After N calls, translate AST to bytecode
  - JVM picks it up and does its thing
- Typically class-per-method

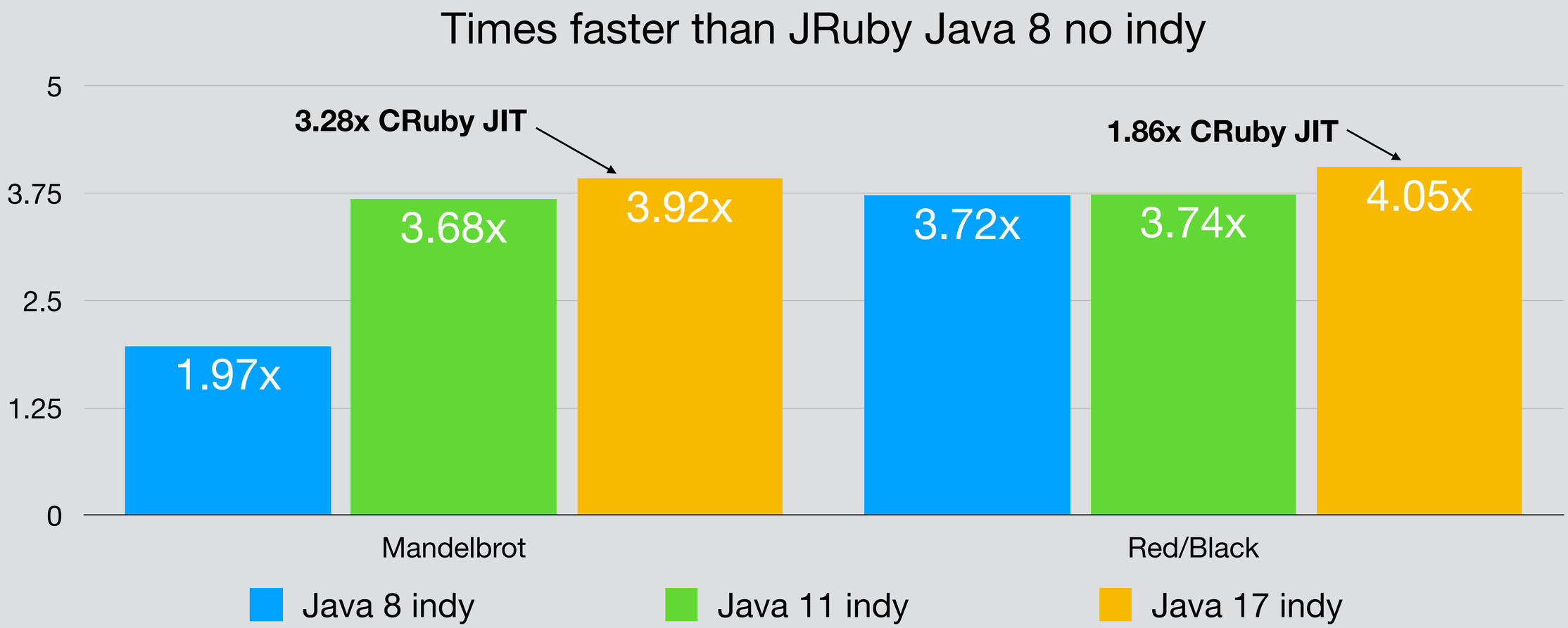
# New Compiler Design

- 2009-2015: Subramaniam Sastry helps us write a new compiler
  - Background in C++ optimizing compilers
- New Intermediate Representation (IR) based on traditional design
  - Register machine with basic blocks, operands, CFA/DFA
- Bytecode JIT got simpler, IR did most of the work

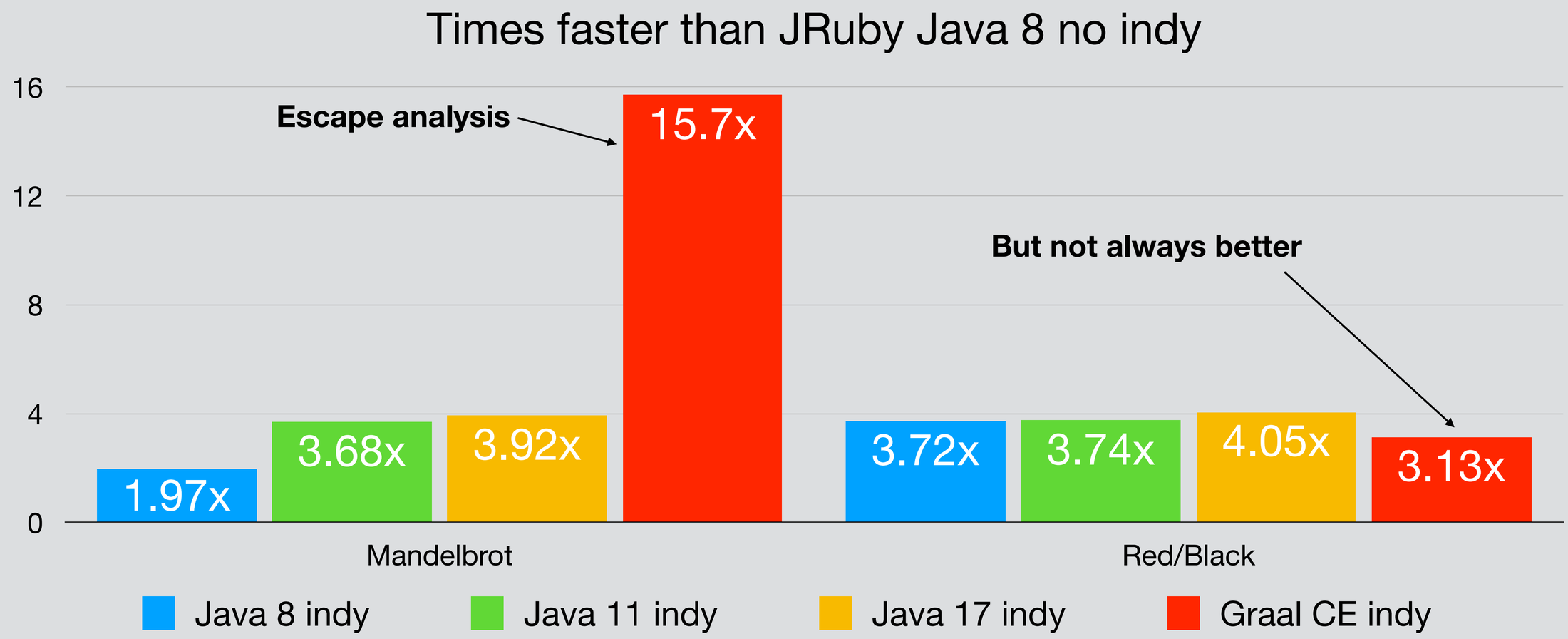
# Invokedynamic

- Introduced in Java 7, steadily improving perf and scaling
- Extensive use throughout JRuby
  - Bytecode is mostly invokedynamic calls
  - "Torture test" for indy performance
- Key to JRuby's higher performance vs CRuby

# Indy on Java 8, 11, 17



# Indy + Graal JIT?



# JVM Today: Dynamic Lang JIT

- Class-per-method is still extremely heavyweight
  - Big part of why we defer compilation
  - Limits our ability to specialize code
- Indy is working very well
  - Tricky to use but we have utility libs that help
- More exotic call sites coming soon

# Native Interop

# Native Interop

- CRuby prefers JNI-like extensions
  - Too invasive, exposing VM guts
- JRuby introduces the Java Native Runtime (JNR)
  - Tools for binding and calling native libraries
  - https://github.com/jnr
- FFI Ruby API ported to JRuby

# Java Native Runtime

- jffi: JNI wrapper around libffi
  - libffi: loading, calling C libraries dynamically
- jnr-ffi: user API for binding and calling C libraries
- jnr-posix: common POSIX functions bound with jnr-ffi
- jnr-enxio: nonblocking native IO bound with jnr-ffi
- jnr-unixsocket: UNIX sockets using jnr-enxio, jnr-ffi
- jnr-process: posix_spawn with selectable native channels

# Ruby FFI

```ruby
class Timeval < FFI::Struct
  layout :tv_sec => :ulong,
         :tv_usec => :ulong
end

module LibC
  extend FFI::Library
  ffi_lib FFI::Library::LIBC
  attach_function :gettimeofday, [ :pointer, :pointer ], :int
end

t = Timeval.new
LibC.gettimeofday(t.pointer, nil)
```

# JVM Today: Native Interop

- Foreign Function and Memory API (Project Panama)
- Independent benchmarks show great promise
  - JNR used to be fastest, now it's Panama
- jextract could finally make FFI easy
  - Currently prototyping Ruby magic to wrap jextract
- JNR backend work already happening! (Michel Trudeau @Oracle)

# jextract + Ruby magic

- Writing bindings using FFI is still challenging
  - Parameter sizes, struct layout, in and out values, pointers
  - Platform differences
- jextract could help generate Ruby FFI calls
  - Same metadata, different wrapper

```
//point.h
struct Point2d {
    double x;
    double y;
};
double distance(struct Point2d);
```

```
import java.lang.foreign.*;
import static org.jextract.point_h.*;
import org.jextract.Point2d;

var session = MemorySession.openConfined();
MemorySegment point =
    MemorySegment.allocateNative(Point2d.$LAYOUT(),
                                 session);
Point2d.x$set(point, 3d);
Point2d.y$set(point, 4d);
distance(point);
```

# SQLite JDBC Adapter

- Java DataBase Connectivity (JDBC) wrapper around SQLite
  - Used by JRuby for ActiveRecord, Sequel
- Java Native Interface (JNI) currently, limits throughput
- Proof-of concept Panama-based version being tested

# Prism: Ruby Language Parser

- Simple C library for parsing Ruby that we can share
- JRuby migrating to it, nearly done
  - Nearly all nodes implemented, 20% faster baseline startup
- Using JNR... Panama faster?
- Also exploring Prism as WASM
  - Run on Chicory when no native lib available (it works!!!)

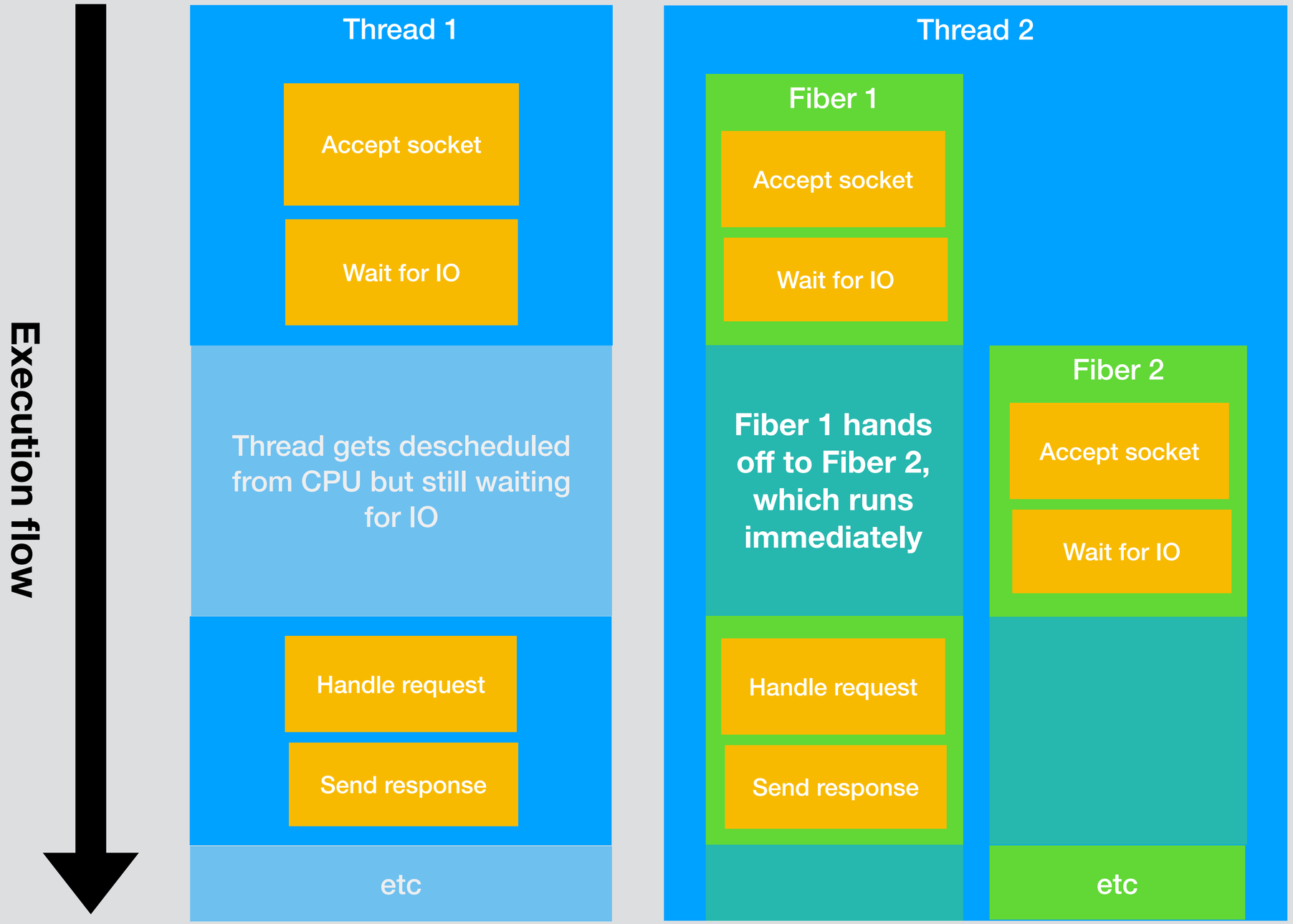
# Lightweight Threads

# Lightweight Threads

- Ruby 1.9 introduced coroutine-like Fiber
  - Single native thread drives several fibers
  - Context switch is voluntary and explicit
- Recently, push toward structured concurrency
  - Auto-reschedule on blocking IO, locks, etc

# Fibers on JRuby

- Without a coroutine API, all were native threads
  - Heavy, limited number, slow context-switch
- Ruby Enumerator#next aggravates this
  - Driving external iterator with internal iteration requires Fiber
- Structured concurrency nearly impossible
  - Thousands of fibers intended for small operations

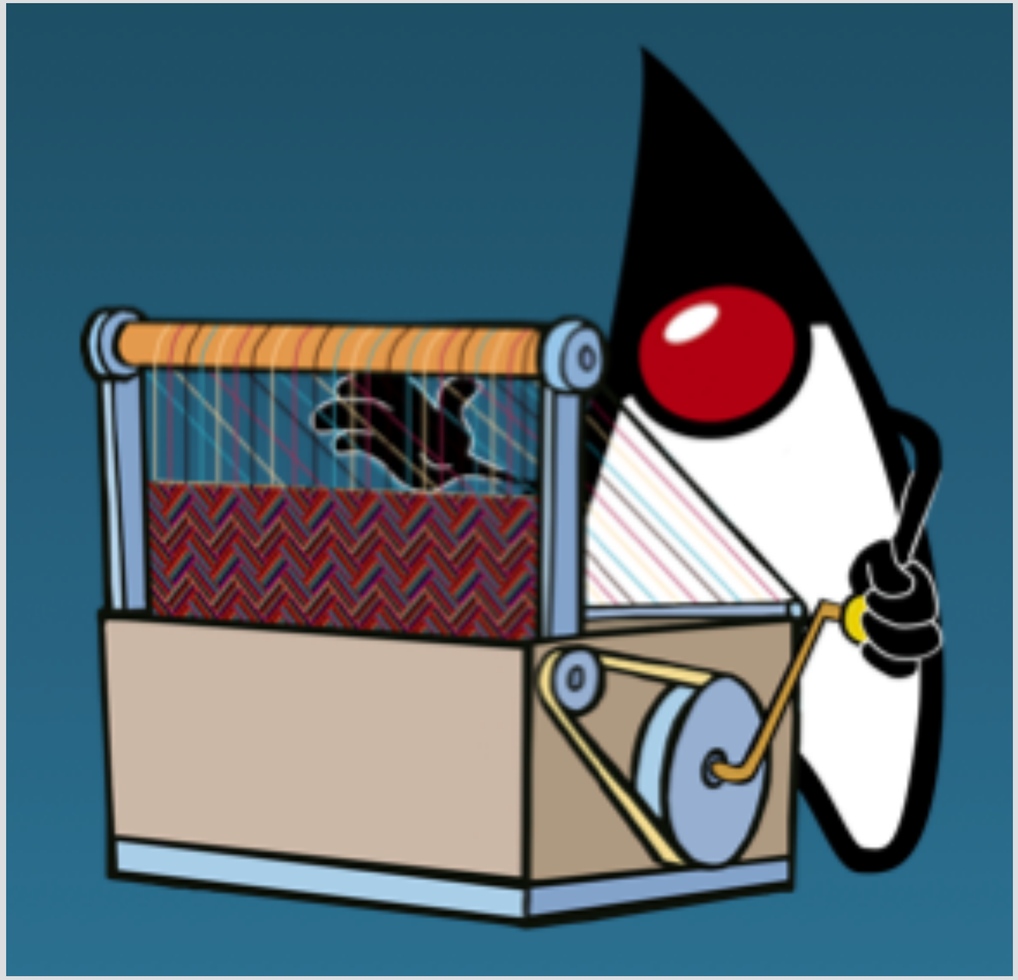

Execution flow
Thread 1
Accept socket
Wait for IO
Thread gets descheduled from CPU but still waiting for IO
Handle request
Send response
etc
Thread 2
Fiber 1
Accept socket
Wait for IO
Fiber 1 hands off to Fiber 2, which runs immediately
Handle request
Send response
Fiber 2
Accept socket
Wait for IO
etc

```ruby
5.times do
  t = Time.now

  # create 100k fibers
  ary = 100_000.times.map { Fiber.new { } }

  # resume and complete 100k fibers
  ary.each(&:resume)

  p Time.now - t
end
```

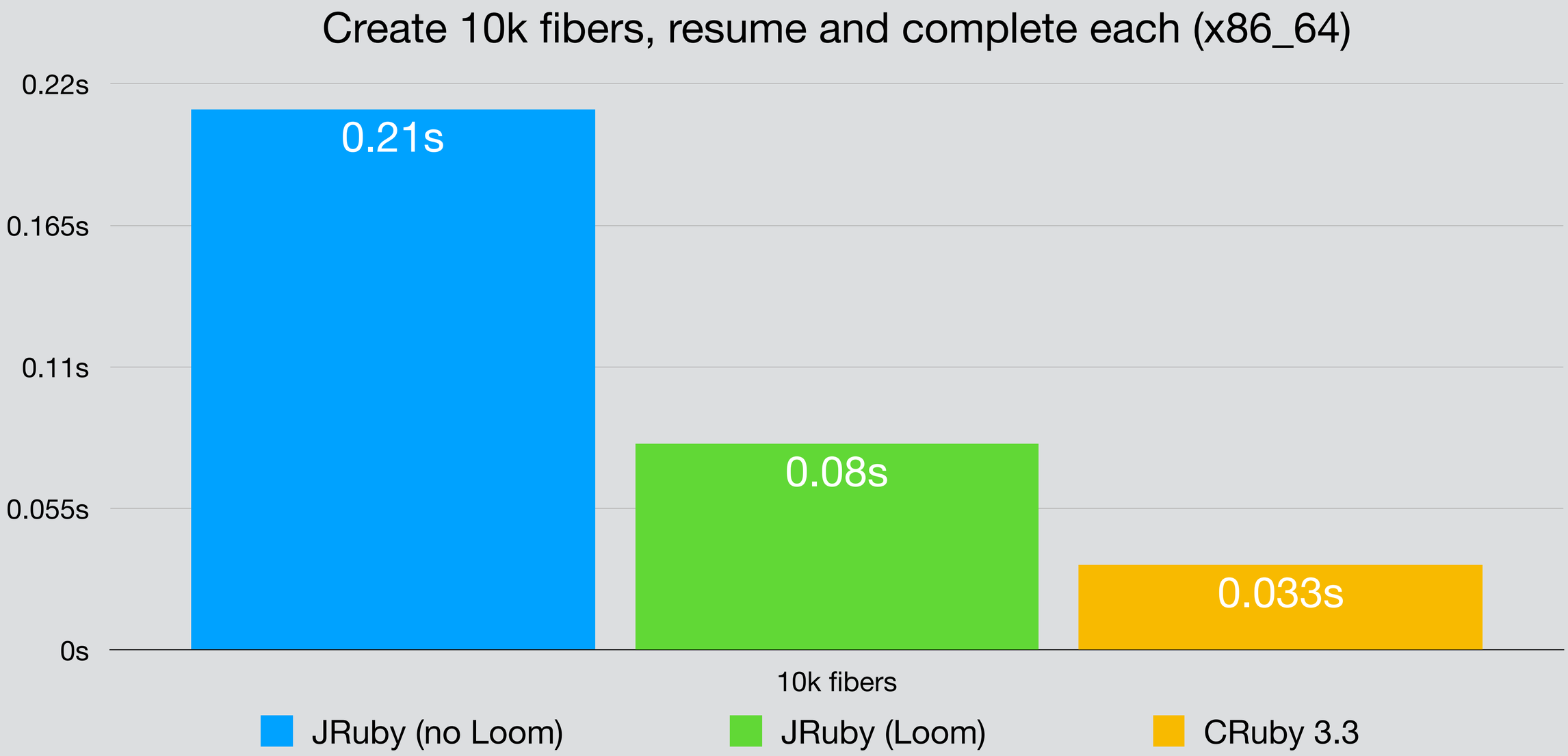
```
$ jruby fiber_test.rb
[7.603s][warning][os,thread] Attempt to protect stack guard pages failed
(0x00007fc240a00000-0x00007fc240a04000).
#
# A fatal error has been detected by the Java Runtime Environment:
# Native memory allocation (mprotect) failed to protect 16384 bytes for
# memory to guard stack pages
#
# An error report file with more information is saved as:
# /home/headius/work/jruby/hs_err_pid75149.log
#
# If you would like to submit a bug report, please visit:
#   https://bugreport.java.com/bugreport/crash.jsp
#
Aborted (core dumped)
```

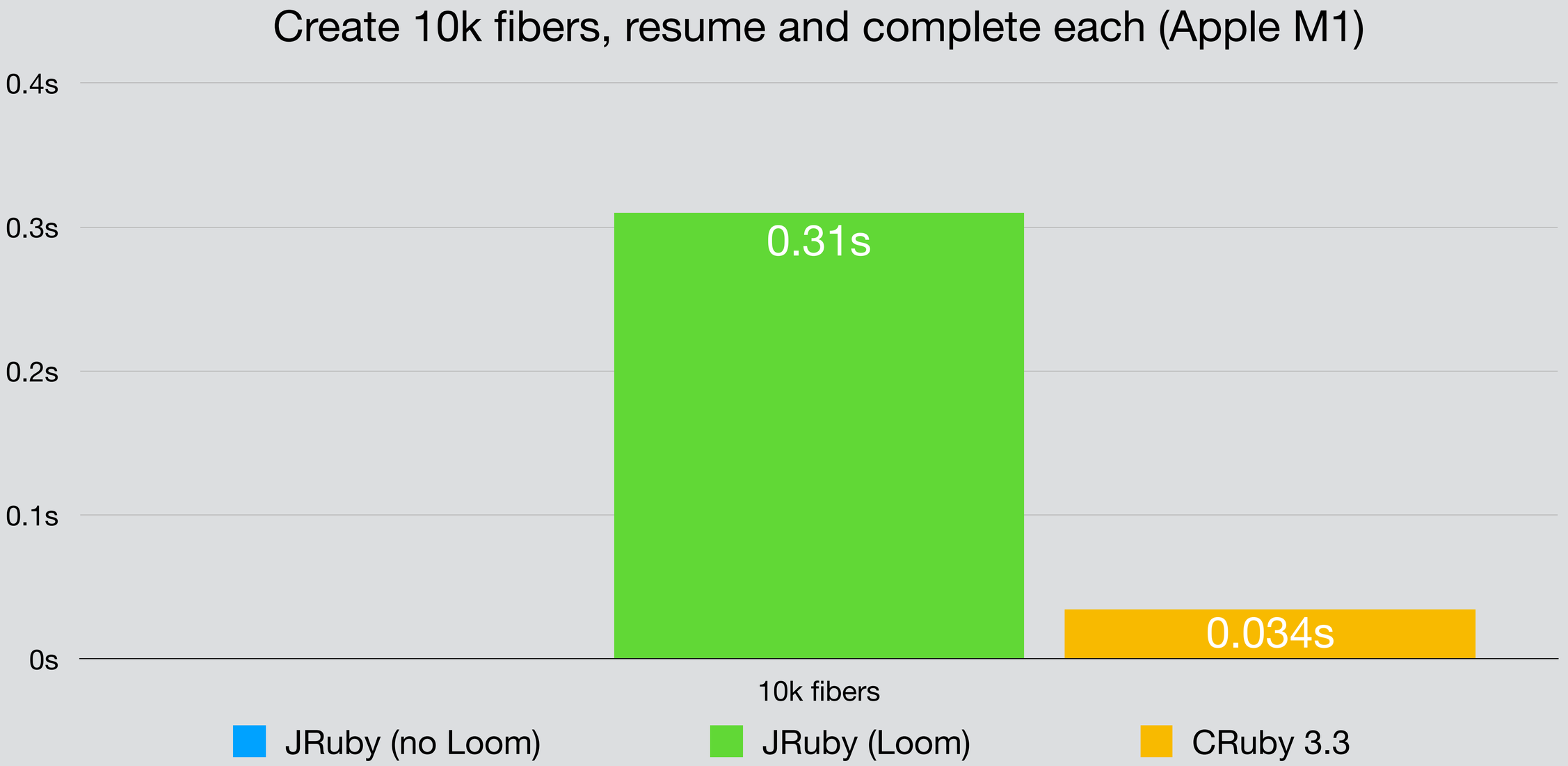
# JVM Today: Virtual Threading

- Project Loom "Virtual threads"
  - User-mode threading
  - JVM handles scheduling
  - Thread does not have to pause
- Perfect analog for Ruby's Fibers
- Already integrated in JRuby



```
diff --git a/core/src/main/java/org/jruby/ext/fiber/ThreadFiber.java b/core/src/main/java/org/
jruby/ext/fiber/ThreadFiber.java
index 6dda30cffa..e6f8803d6b 100644
--- a/core/src/main/java/org/jruby/ext/fiber/ThreadFiber.java
+++ b/core/src/main/java/org/jruby/ext/fiber/ThreadFiber.java
@@ -281,7 +281,7 @@ public class ThreadFiber extends RubyObject implements ExecutionContext {

         while (!retried) {
             try {
-                runtime.getFiberExecutor().execute(() -> {
+                Thread.ofVirtual().start(() -> {
                     ThreadContext context = runtime.getCurrentContext();
                     context.setFiber(data.fiber.get());
                     context.useRecursionGuardsFrom(data.parent.getContext());
```


Create 10k fibers, resume and complete each (x86_64)
0.22s
0.165s
0.11s
0.055s
0s
0.21s
0.08s
0.033s
10k fibers
JRuby (no Loom)
JRuby (Loom)
CRuby 3.3

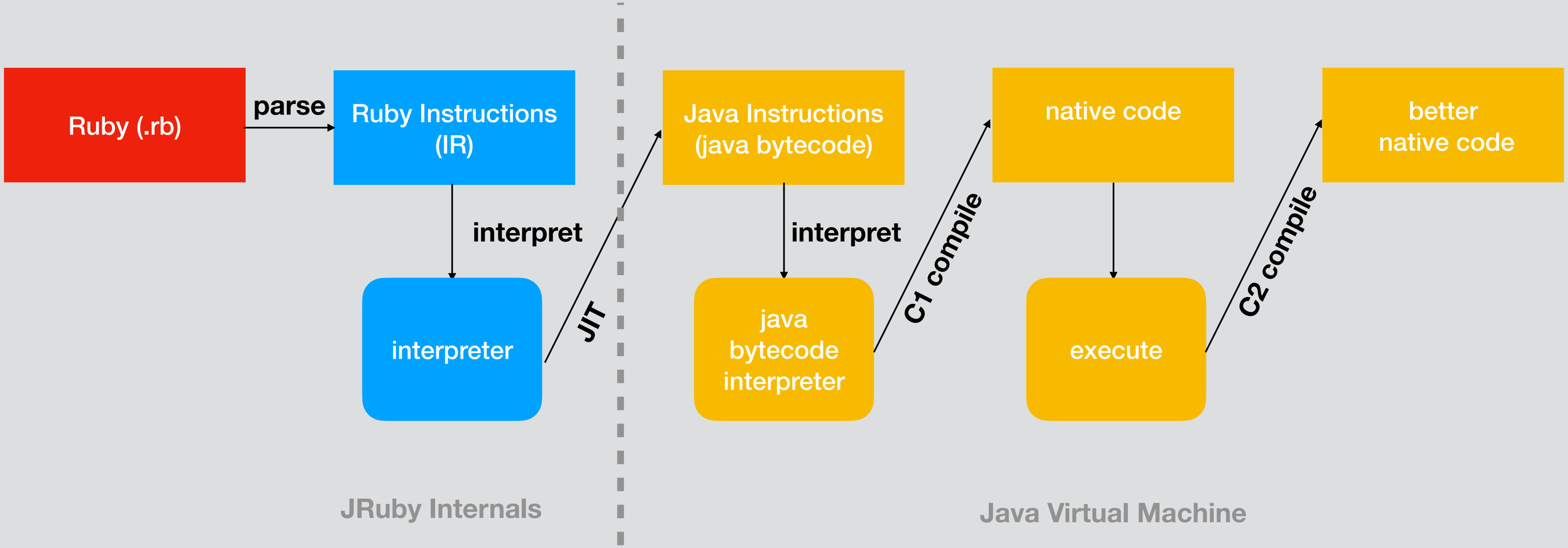

Create 10k fibers, resume and complete each (Apple M1)
0.4s
0.3s
0.2s
0.1s
0s
0.31s
0.034s
10k fibers
JRuby (no Loom)
JRuby (Loom)
CRuby 3.3

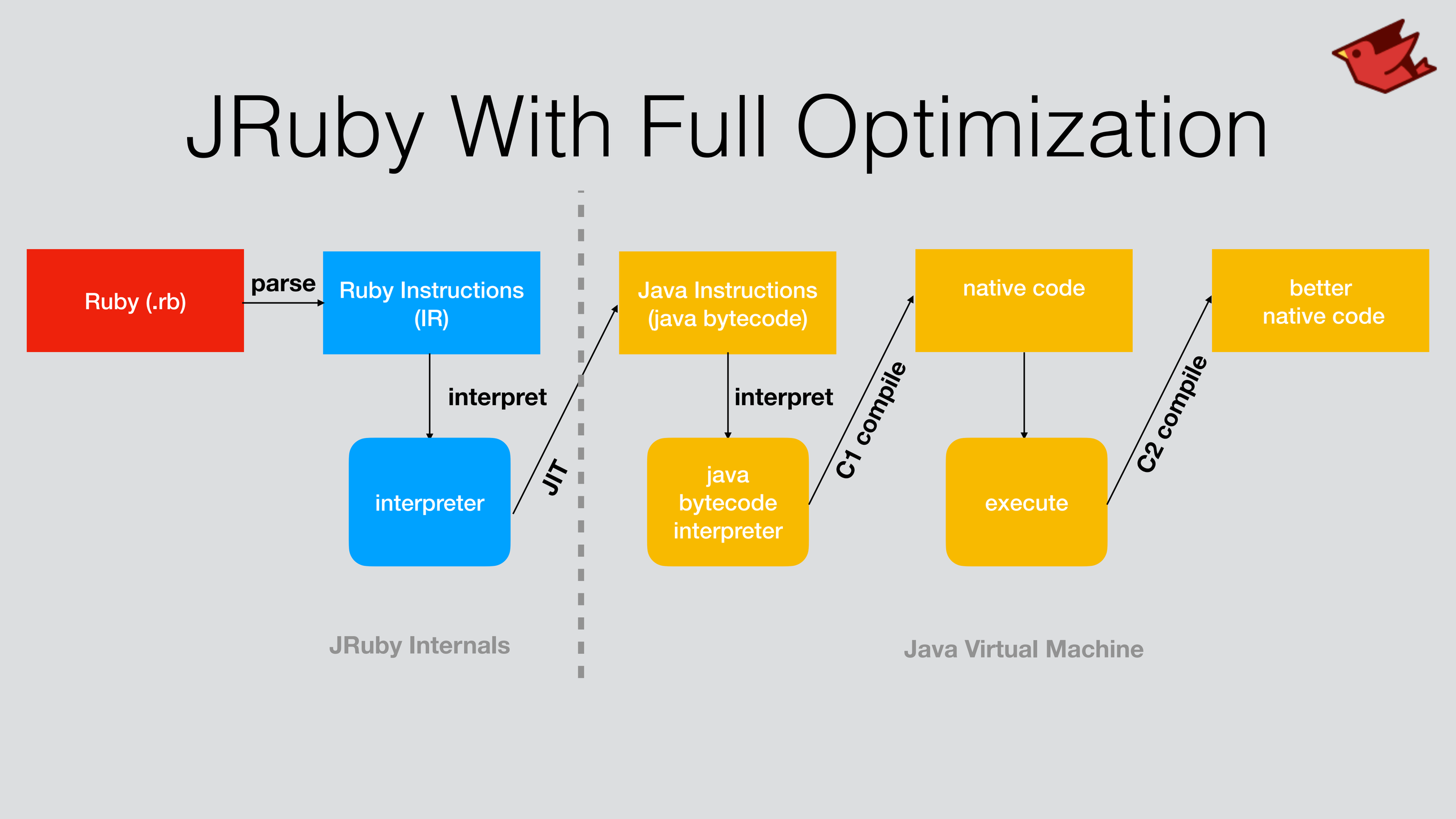
# Startup and Warmup

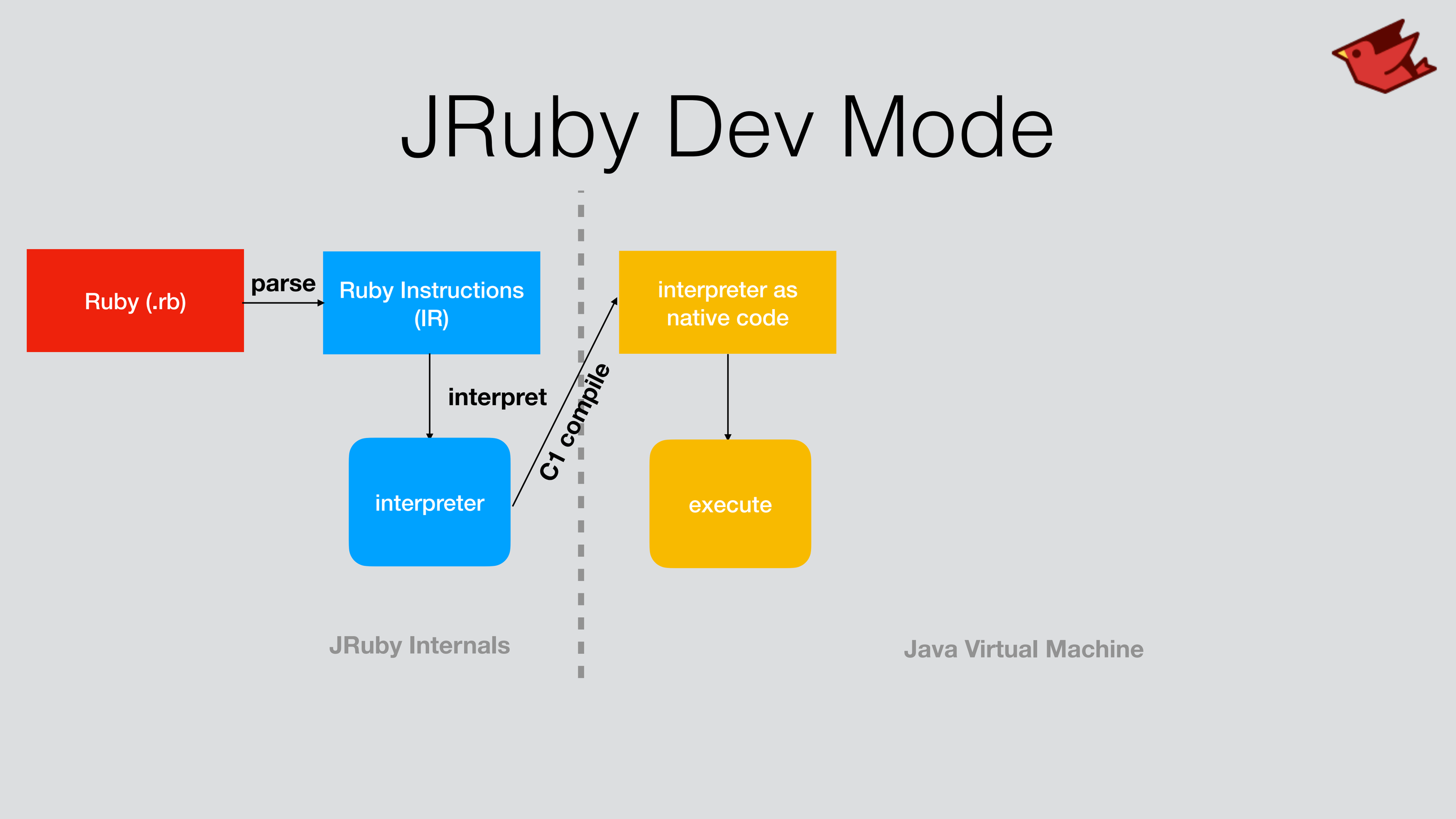
# Startup and Warmup

- JVM is not designed to start up quickly
  - Most of core JDK starts in interpreter
  - Long tail to optimize code and reach peak performance
- JRuby makes it harder
  - Interpreting Ruby initially, interpreter is interpreted by JVM
  - Lazy compile to bytecode, bytecode is interpreted by JVM

# JRuby Architecture




JRuby With Full Optimization
Ruby (.rb)
parse
Ruby Instructions
(IR)
interpret
interpreter
JIT
Java Instructions
(java bytecode)
interpret
java
bytecode
interpreter
C1 compile
native code
execute
C2 compile
better
native code
JRuby Internals
Java Virtual Machine



# JRuby Dev Mode

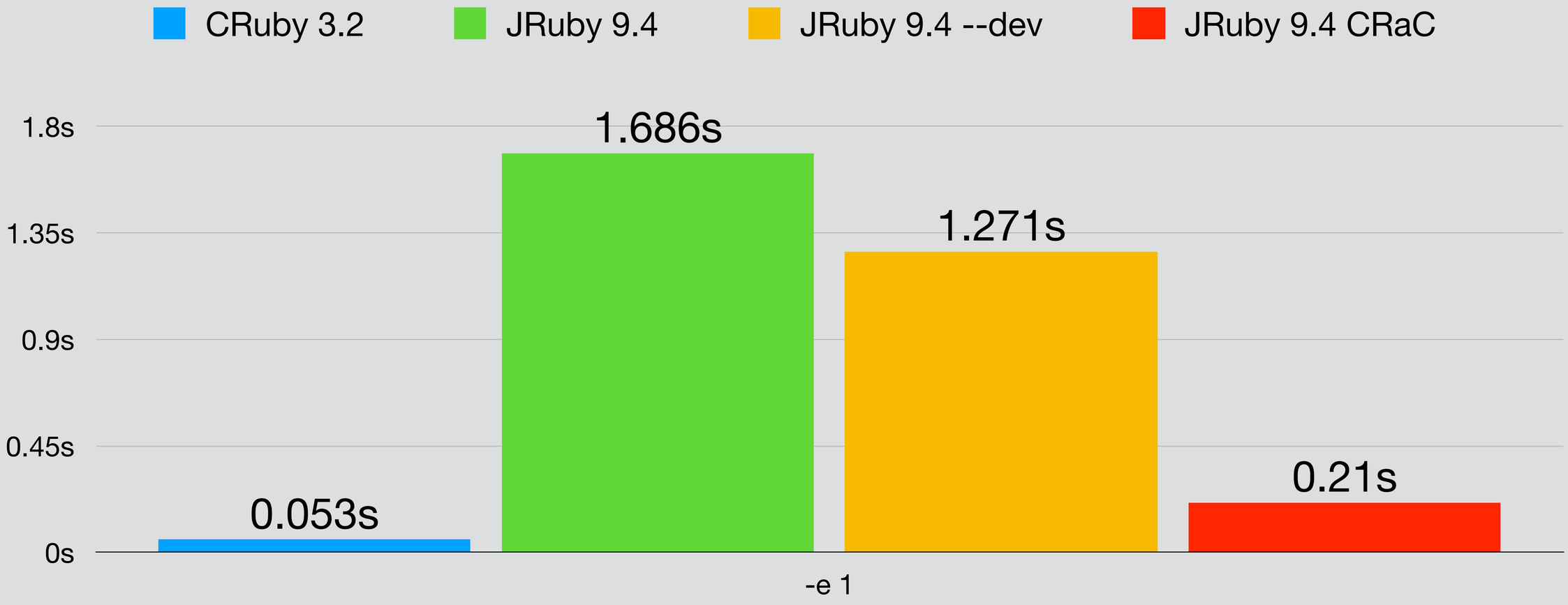
# Ahead-of-time Compilation?

- Maybe we can start with native code?
- GraalVM Native Image is a well known option
  - But completely disables dynamic features of JVM!
- Project Leyden hopefully works better?

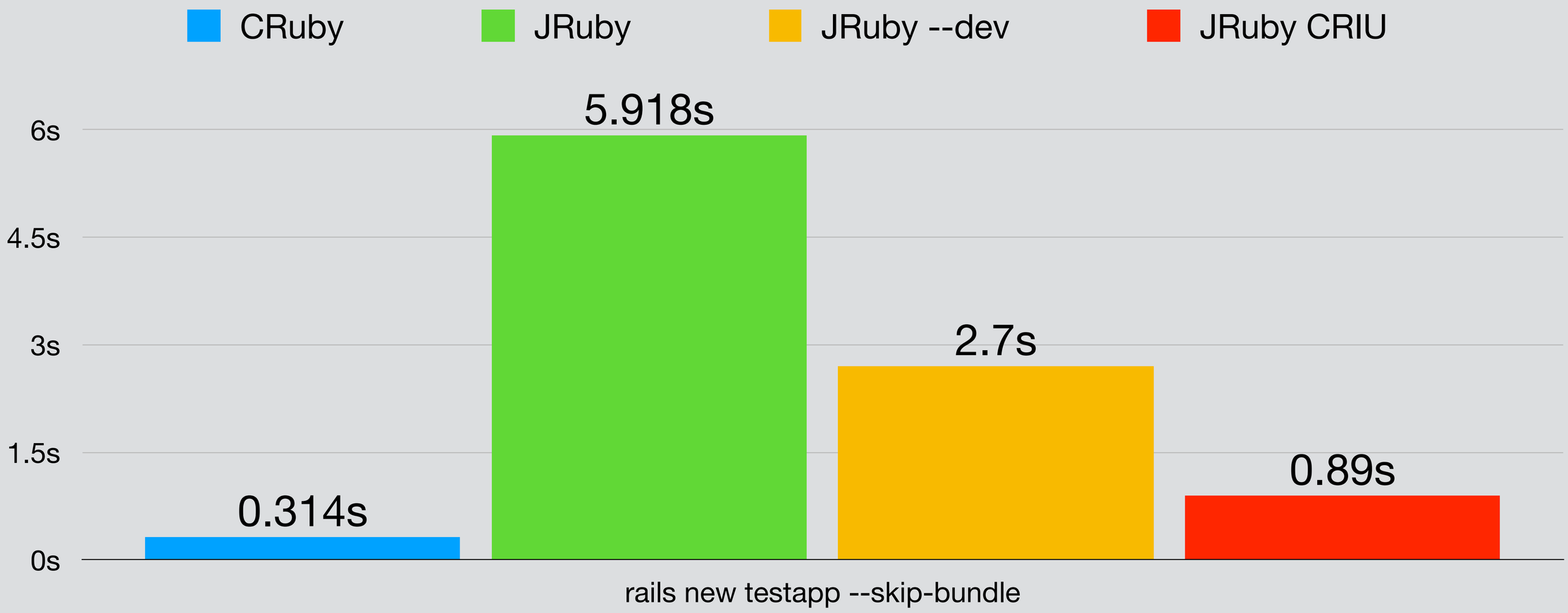
# JVM Today: Startup/Warmup

- Checkpointing most likely solution short-term
  - CRIU: Checkpoint and Restore in Userspace
  - CRaC: OpenJDK support for checkpoint and restore
  - Cross-platform possible?
- We want Leyden to work, willing to help
  - JRuby requires bytecode JIT, invokedynamic today

# ruby -e 1


CRuby 3.2
JRuby 9.4
JRuby 9.4 --dev
JRuby 9.4 CRaC
1.8s
1.35s
0.9s
0.45s
0s
0.053s
1.686s
1.271s
0.21s
-e 1

# rails new testapp --skip-bundle



# JRuby is a Testbed

- No other JVM language pushes so many edges
  - Future JVM languages may look similar
- Large corpus of tests, benchmarks from Ruby world
- Real-world users willing to run leading edge
- Always new challenges for JRuby and JVM!

# More Background

- The Java Native Runtime (JVMLS 2013, FOSDEM 2014)
- JRuby: The Hard Parts (JVMLS 2014)
- Invokedynamic: Tales from the Trenches (FOSDEM 2013)
- Optimizing Above the JVM in JRuby 9000 (FOSDEM 2016)
- Ruby's Strings and What Java Can Learn From Them (FOSDEM 2017)
- JRuby Startup and AOT (FOSDEM 2020)

# Thank You!

- Charles Oliver Nutter
  - headius@headius.com
  - @headius(@mastodon.social)
- https://github.com/jruby/jruby
- https://www.jruby.org