JRuby on Graal

Performance and Startup Experiments
JRuby Review

- Ruby for the JVM
  - Two-way integration with Java, fitting into ecosystem
  - We are a Ruby implementation, but also a JVM language
- Core classes largely written in Java
- Parts of core and most of standard library in Ruby
- Distribution like CRuby or as jars/wars, embedded into apps
- No support for CRuby extensions, on purpose
JRuby Architecture

Ruby (.rb) → parse → Ruby Instructions (IR) → interpret → interpreter

Java Virtual Machine

Java Instructions (java bytecode) → interpret → java bytecode interpreter

native code → C1 compile → better native code

deoptimize
JRuby Challenges

• Java bytecode is a narrow vocabulary
  • InvokeDynamic helps but adds complexity
• Object boxes are too expensive
• Lambda-style code optimizes poorly
• Startup time, memory footprint are crucial for adoption
• Two FTEs barely keeps up with compatibility, user issues
JRuby and Graal
History

- Experimented with Maxine back in the day
- Collaborated with TruffleRuby early on
- Investigating JRuby performance on Graal
  - Playing with compiler passes
  - Studying compiler IR, assembly code for opportunities
Today

- JRuby on Graal straight-line performance
  - Microbenchmarks up to small web services
- JRuby native with GraalVM
  - Working POC
- Plans going forward
Performance
It's a Hard Problem

- Heavy use of invokedynamic
  - Method calls, constants, globals, instance variables, ...
- Limited specialization
  - Object shaping, flattened arrays, frame elimination, splitting
- Looking for new opportunities
  - e.g. "truly final" final fields
General Notes

- Java 8, Java 13, GraalVM 20
  - Invokedynamic, fixnum caching options
  - Java 13 using -XX:+UseParallelGC
- Iterations or requests per second (higher is better)
- Force compilation to JVM bytecode (no interpreted phase)
Integer Loop

- Simple while loop from zero to 10M
- "nanobenchmark"
- Small method, simple integer math, conditional looping
Integer Loop

- **Java 8**: 12.5
- **Java 8 + Indy**: 18
- **Java 13 + Indy**: 20.3
- **Graal 20 + Indy**: 22.5

Iterations per second
Helping Graal

- Make more state final
- Fewer loads, more constant propagation
- Avoid caching elidable objects
  - Mixing real and virtual objects seems to cause problems
  - Added a flag to disable Fixnum cache (like Integer.valueOf)
Integer Loop

Iterations per second

Java 8
Java 8 + Indy
Java 13 + Indy
Graal 20 + Indy
Graal 20 + Indy - Fixnums

12.5
18
20.3
22.5
67.1
Mandelbrot

- Microbenchmark: one moderately-sized method
- Nearly all numeric computation
  - Reasonable baseline for numeric algorithm performance
- Worst case for JRuby on most JVMs
  - 100% boxed numerics
  - Allocation rather than GC is the bottleneck
Mandelbrot Optimizations

- Final references to Boolean objects, core classes
- Keep literal numerics as primitives
- Avoid caching Fixnum objects
Optcarrot

- Nintendo Entertainment System emulator in pure Ruby
- Heavy use of simulated memory (integer arrays), dynamic dispatch
- Very little optimization work on JRuby side
Optcarrot

First 100 iterations

- JRuby Java 13
- JRuby Graal 20
- Ruby 2.6.5 JIT
Optcarrot

Last 100 iterations

- JRuby Java 13
- JRuby Graal 20
- Ruby 2.6.5 JIT
Applications

- Roda
  - Microservice-style web framework
- Rails
  - Heavily dependent on ActiveRecord performance
- CRuby vs JRuby, JRuby + Graal
Roda

- Small, well-supported service framework
- Many production users at large scales
- Very simple example with no database
- Benchmarking request routing mostly
- Good indicator of small app performance
Setup

- CRuby: 8 processes
- JRuby: 8 threads
- Driver: wrk with 16 connections, 2 reactor threads
Roda Full Concurrency

- CRuby
- JRuby Java 13
- JRuby Graal 20
- JRuby Java 13 Graal
ActiveRecord Performance

- Rails apps live and die by ActiveRecord
  - Largest CPU consumer by far
  - Heavy object churn, GC overhead
- Create, read, and update measurements
- CRuby 2.5.1 vs JRuby 9.2 on JDK11
ActiveRecord Selects

time for 1000 selects, lower is better

- CRuby 2.5
- JRuby C2
- JRuby Graal

- binary
- boolean
- date
- datetime
- decimal
- float
- integer
- string
- text
- time
- timestamp
JRuby + Graal

• Clear wins for small, object-heavy benchmarks
• Larger applications are a mixed bag
  • Need to dig deeper and see why
• Potential to be the fastest way to run JRuby
  • Applicable to other languages and libraries on JVM
Startup
JRuby Architecture

Ruby (.rb) → Ruby Instructions (IR) → interpreter

Java Instructions (java bytecode) → java bytecode interpreter

native code → execute

better native code

deoptimize

JRuby Internals

Java Virtual Machine
total execution time (lower is better)

- CRuby
- JRuby (JDK8)
- JRuby (10th iter)
- JRuby (50th iter)

1.63
0.124
0.075
0.1
Startup Experiments

- Preboot or reuse JVM process
- Save parse results, compiled IR
- Precompile to native
GraalVM Native Image

- Compile all of JRuby to native (working POC)
  - Build times in 2-3min range... not bad
- Many limitations
  - No invokedynamic, limited reflection, no dynamic classloading, ...
- Eventual goal: fully native Ruby apps (no startup or warmup)
  - Compile Ruby to bytecode, and then to native
  - Good for tools, microservices
CRuby | JRuby (JDK8) | JRuby native
---|---|---
0.1s | 1.63s | 0.117s

total execution time (lower is better)
Bytecode AOT Mode

- AOT mode: No indy at all
- A bit more bytecode generated
- Only direct method handles or LambdaMetaFactory objects
- Cold bytecodes reduced vs normal precompile
Next Steps

- Compile Ruby app + library sources to native
  - Needed bytecode AOT to proceed
- Static optimizations
- Remove unneeded parts of JRuby
- Probably limited to small services, command line tools
  - libjruby?
Thank you!

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