Tartu University

Faculty of Social Studies

Narva College

Study program “Information Technology Systems Development”

Andrei Kazakov

COMPARISON OF PERFORMANCE OF RUBY INTERPRETERS IN HIGH-LOAD RAILS APPLICATIONS

Bachelor thesis

Supervisor: Andre Sääsk, M.Sc.

Narva 2023

DECLARATION

Olen koostanud töö iseseisvalt. Kõik töö koostamisel kasutatud teiste autorite tööd, põhimõttelised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

/allkirjastatud digitaalselt/
Andrei Kazakov
12.05.2023

ABSTRACT

Selle töö teema on Ruby interpretaatorite jõudluse võrdlus suure koormusega Rails-rakendustes. See uurimus on relevantne firmadele, mille IT-süsteemid on kirjutatud Ruby programmeerimiskeeles ja töötavad suure koormuse all. Selle töö uurimisprobleem on aktuaalne kaasaegses IT-valdkonnas, kuna GitHub-i andmetel Ruby on kümnes kõige populaarsematest programmeerimiskeeltest 2022. aasta seisuga.

Töös käsitletakse madal Ruby programmeerimiskeele jõudlus suure koormusega IT-süsteemides ja püstitatakse uurimisküsimus, kas interpretaatori vahetamine võib suurendada veebirakenduste jõudlust. Selle küsimuse vastamiseks mõõdetakse veebirakenduse jõudlus erinevate interpretaatorite all, modeleerides üht mikroteenust pilvekeskkonnas.

Selle töö peamiseks järelduseks on see, et alternatiivse interpretaatori kasutamine toob kaasa kuni 94-protsendilist jõudluse parendust ja JVM-põhised interpretaatorid on jõudluse suhtes parimad. Järelduseks on ka see, et jõudluse parendus toob kaasa ka kompromisse, mis tähendab, et interpretaatori vahetamine olemasolevas IT-süsteemis vajab ka eraldi katsetamist.

TABLE OF CONTENTS

ABBREVIATIONS AND DEFINITIONS

AMI — Amazon Machine Image

AOT compilation — ahead-of-time compilation; an optimisation technique

API — Application Programming Interface

AWS — Amazon Web Services

CDN — Content Delivery Network

CPU — Central Processing Unit

CRuby — the official Ruby interpreter implemented in C programming language

GC — Garbage Collector

GIL — Global Interpreter Lock; sometimes referenced as GVL (Global VM Lock)

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

IO operations — Input-Output operations

IPO — Initial Public Offering

JIT compilation — just-in-time compilation; an optimisation technique

JRuby — Ruby interpreter implemented in Java and running on JVM

JVM — Java Virtual Machine

LTS — Long-Term Support

MJIT — Method-based JIT compiler in CRuby

MRI — Matz's Ruby Interpreter, the original implementation of the Ruby language interpreter before switch to YARV in CRuby 1.9

MSL — Maximum Segment Lifetime in TCP specification

p50 — 50-th percentile, the same as median; 50% of the values in the original set are equal or less than p50 value

p99 — 99-th percentile; 99% of the values in the original set are equal or less than p99 value

RPS — requests per second

RRB — Rails Ruby Bench; a benchmarking suite using Discourse as the benchmark application

TCP — Transmission Control Protocol

vCPU — a virtual CPU provided by the virtualisation environment

VM — Virtual Machine

YARV — Yet Another Ruby VM, the official Ruby interpreter since CRuby 1.9

YJIT — JIT compiler in CRuby based on a Lazy Basic Block Versioning (LBBV) architecture

INTRODUCTION

BACKGROUND

The Ruby programming language was first publicly released in 1995, and at the moment of writing Ruby is the 10th most popular programming language on GitHub (‘The Top Programming Languages’, 2022). Developers consider it beautiful, artful, flexible and expressive, as this language tries to balance functional and imperative approaches to programming and its author, Yukihiro “Matz” Matsumoto, tries to “make Ruby natural, not simple” (‘About Ruby’, n.d.).

Most of its current popularity comes from the popularity of Ruby on Rails framework. A lot of well-known companies use applications written in Ruby on Rails as an important part of their high-load web stacks (Egeonu, 2022). It makes it easy to start the project from scratch and then scale it up, and the official website says “Ruby on Rails scales from HELLO WORLD to IPO” (‘Ruby on Rails’, n.d.). “It is designed to make programming web applications easier by making assumptions about what every developer needs to get started. It allows you to write less code while accomplishing more than many other languages and frameworks. … Rails is opinionated software. It makes the assumption that there is a “best” way to do things, and it is designed to encourage that way — and in some cases to discourage alternatives.” (‘Getting Started with Rails’, n.d.)

While there are a lot of advantages of Ruby on Rails, such as abundance of tools and libraries, extensive use of tests, higher third-party code quality, large and responsible community and high development productivity, there’s one very often cited disadvantage of this framework — it is slow, which mainly refers to slow runtime, but sometimes to slow boot speed and not leveraging concurrency features offered by libraries and interpreters (Ganguly, 2018).

RESEARCH QUESTION

Our research focuses on the slow runtime aspect, as the slower a web application processes incoming requests, the less amount of requests it will be able to process using the same amount of server resources. Speeding up the runtime will help companies to reduce infrastructure costs while serving the same amount of requests, or serve increased amounts of requests with the same costs.

Our hypothesis is that different Ruby interpreters may show different runtime performance speed. So, the research question is:

Can alternative Ruby interpreters show increased performance when used in a high-load web backend?

PREVIOUS WORK

Ruby was designed as a “beautiful and expressive” language, which, basically, contradicts the idea of high performance, as “expressive” features are extremely hard to implement in an efficient way (Bini, 2008). At the same time — especially with the appearance of the Rails framework — developers liked the expressiveness of the language, which helped to build applications faster (more efficiently in terms of personal workload per software engineer). These applications started to become more and more successful (in business terms) and experienced more and more workload (in requests per second), which highlighted the slowness of Ruby code at all and Ruby interpreters in particular.

While there are a lot ot ways to optimise the application itself, such as code optimisation (Dymo, 2015), caching, disabling redundant modules, profiling, optimising database queries and reducing the amount of them (Klochkov & Mulawka, 2021), such optimisations will be out of scope of this research, as I’ll be trying to run exactly the same code across all the interpreters during our research. Performance issues of the interpreters themselves, of course, have been addressed a lot too.

One of the first attempts to address the slowness of the official Ruby interpreter was the invention of YARV (Sasada, 2005), which, due to its optimisations, was 16x more efficient in

some benchmarks than the previous interpreter implementation — MRI. YARV also switched from the non-efficient user-level “green thread” concurrency model to using native system threads, while still not supporting true parallelism because of using GIL. Sasada also planned to add AOT compilation to the interpreter, but had no plans to introduce JIT compilation into YARV and considered it “not reasonable (not worth the cost of implementation)” (Matsumoto & Sasada, 2007). YARV replaced MRI as the default Ruby interpreter in 2007, starting from CRuby 1.9 (CRuby — as the default Ruby implementation — still may be called MRI, which sometimes leads to confusion).

In 2015, Matz announced the “Ruby 3x3” initiative with a goal to make CRuby 3.0 three times faster than CRuby 2.0 (Matsumoto et al., 2016). The actual work on improving the official interpreter's performance has started much earlier: for example, GC optimisations started in CRuby 2.1 in 2013 and continued onwards (Sasada, 2019). There was a lot of work to introduce JIT compilation into CRuby, like MJIT in CRuby starting from 2.6 (Matsumoto et al., 2018) and YJIT in CRuby staging from 3.1 (Chevalier-Boisvert et al., 2021). A lot of other optimisations have been done too.

Alternative Ruby implementations, such as JRuby, also have been existing for a long time. Speaking of JRuby, its developers focused on leveraging JVM advantages (such as true native-thread parallelism without GIL, efficient GC and other optimisations) as well as AOT and JIT compilation techniques, which allowed it to beat MRI in various benchmarks at the very beginning of its development (Cangiano, 2008).

The examples demonstrated by Cangiano in 2008 (comparison of CRuby 1.8.7, CRuby 1.9.1, Ruby Enterprise Edition, JRuby and Rubinius) clearly show that due to their different history of development, evolution, approaches and underlying technologies, different Ruby interpreters may show significant difference in performance. A promising research task would be to repeat such a comparison in a modern, cloud-based environment with modern Ruby interpreters.

RELEVANCE

Results of our research are relevant for companies whose IT systems experience high load, serve HTTP requests and are written in Ruby using Rails, and also for Ruby developers and enthusiasts.

LIMITATIONS

Out of scope of our research are:

- Optimising application code, database queries or the interpreter itself
- Measuring the performance of interpreters not installable using ruby-build
- Measuring the application's memory footprint or IO usage
- Measuring the performance of different application's endpoints
- Measuring the performance of interpreters not being able to run the benchmark application
- Measuring the impact on Rails assets, templates or static files rendering
- Measuring the impact of changing the application server
- Measuring the impact of environment changes, like replacing the database server or changing network configuration
- Measuring the impact of caching, scaling and other architecture-level optimisation techniques
- Measuring the performance in non-cloud or non-microservice environment
- Measuring the impact of hardware or cloud platform change, including the change of processor architecture
- Measuring the impact of changing application server settings.

Many of these points may be addressed in future research works.

The present research is focused only on measuring the performance of an API-only Rails application in the cloud. While this experiment design shows relative differences between different interpreters in such circumstances, this does not mean that the results will be the same for every Rails application, especially in production conditions and with real workload.

It is also worth keeping in mind that some of the interpreters mentioned in this research may be not production-ready yet, or even just be someone's "pet project" with no intention to make any stable releases. This research provides some insight into performance differences and helps to learn more about Ruby interpreters and their evolution, but it is not an action guide: interpreter replacement in a company's IT systems is likely a very complicated process, which should be approached extremely carefully and with prior investigation and experimentation.

It is also worth keeping in mind that Ruby interpreters are in active development — for example, a new CRuby version is released every Christmas ('Ruby Version Policy Changes Starting with Ruby 2.1.0', n.d.), which means that the measurements in this research may become obsolete very quickly.

1. PREPARING THE BENCHMARK

1.1. ARCHITECTURE OF HIGH-LOAD WEB APPLICATIONS

The best-practice approach to the architecture of a modern high-load web application is the microservice architecture.

While applications with traditional monolithic architecture can still be successful, companies often try to decompose their monolithic applications into microservices, that is to build their applications as suites of services. “As well as the fact that services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages. They can also be managed by different teams.” (Fowler & Lewis, 2014)

The advantages of microservice architecture as maintainability, reusability, scalability, availability and automated deployment are well-recognised across the industry. “Netflix, Amazon and eBay all have migrated their applications from monolithic architecture to microservices architecture, so as to enjoy the advantages that microservices architecture brings about. For example, Netflix, which is a famous video streaming service, can deal with one billion calls every day now by its streaming API in microservices architecture.” (Chen et al., 2017)

While Chen et al. also suggest that microservice architecture is not a panacea and it is challenging for the companies to decompose their monolithic applications into smaller services, it is proven that in addition to the benefits mentioned before microservice architecture may also significantly reduce infrastructure costs (Villamizar et al., 2017).

For purposes of our research, this means that the benchmark application should represent a single service within a microservice setup.

1.2. BENCHMARK APPLICATION

1.2.1. Non-functional requirements

- No sophisticated networking capabilities (HTTPS, slow client support or load balancing) are needed at the level of the benchmark application server, as it is supposed to model a single service within a microservice setup, which means that all directly served clients will be within the same local network. In a real world scenario, these clients will usually be reverse proxies and load balancers (in case we expose the API to the internet) or other microservices within the same setup. In our experiment scenario, the only client was the benchmarking tool.
- The web server should be puma (puma, 2011/2023). It is the default web server bundled with Rails. It has been specifically designed for handling a lot of requests in parallel and it also outperforms other solutions in almost all types of workloads (Haynes, 2019; Pavese, 2016).
- As for the Rails version, we took the oldest maintained version. It should be old because we aimed to give a chance to even exotic Ruby interpreters to run it; it should be maintained because we intended for our research to be relevant to the industry, and companies tend not to keep unmaintained code in production. The oldest maintained version of Rails right after defending the thesis is expected to be 6.1, as 6.0 is discontinued since 01.06.2023 (‘Maintenance Policy for Ruby on Rails’, n.d.).
- It is desirable, although not necessary, to use the same benchmark application which previous researchers have already used in their performance comparisons to achieve comparability of obtained results with other works.

1.2.2. Functional requirements

The main functional requirement for the benchmark application is to simulate the workload which is close to what a real-world application would make in production. In our experience, production applications mostly do IO operations on the network to access databases (which themselves often turn out to be the performance bottleneck (McWherter et al., 2004)), key-value storages, caches, log and telemetry collection systems and other external APIs.

They rarely do anything CPU-heavy; they often do not even have to render page templates, as modern high-load backends are API-only (as they are mostly supposed to work with SPA or mobile application frontends). Rails by default also does not serve static files in production, as it assumes that it is delegated to a separate web server or CDN (RailsGuides, n.d.).

While sometimes it makes sense to also measure CPU-heavy workloads in Rails applications to research differences in concurrency implementation (Pavese, 2016), this is not the case for our research, as we attempted to measure performance of interpreters in close to real-life conditions, which are IO-heavy.

1.2.3. Choosing the benchmark application

After the analysis of previous studies, we have chosen two benchmark applications that correspond with these requirements to the most extent:

- railsbench (Kokubun, 2019/2023) is basically an auto-generated (using “rails generate scaffold” command) very simple application based on Rails 6.1.6.1
- Discourse (discourse, 2013/2023) is an open source discussion platform, which version 2.8.13 is based on Rails 6.1.6.1. Sometimes it might be referenced as Rails Ruby Bench or RRB, as it is the name for the corresponding benchmarking tool (Gibbs, 2016/2023).

Discourse is a complex Rails application actually used in real-world scenarios, which means that choosing this application as a benchmark will emulate the real workload on the server close enough. It is also IO-heavy, as it depends on PostgreSQL and Redis servers. However, compared to railsbench, it also has some disadvantages:

- Since we have not participated in its development and the code is very complicated to be quickly understood, it may introduce some unexpected factors into the benchmarking process
- It may not run with some exotic interpreters, as there is likely more Ruby features that it is going to use in comparison to a simple auto-generated Rails app

- It is a monolithic and complicated application which contradicts the idea of benchmarking a single simple service within a microservice setup
- It is not an API-only application, which, again, contradicts the idea of benchmarking an API-only microservice.

Considering all that, we have chosen railsbench as the benchmark application.

2. SELECTING SUITABLE INTERPRETERS

2.1. REFERENCE IMPLEMENTATION

An interpreter is a program which defines and implements a programming language (Reynolds, 1972). This poses a problem, as different interpreters may implement different definitions of the Ruby language.

As there currently exists a plethora of different Ruby language implementations under active development (codicoscepticos, 2012/2023), this poses a question of determining what the reference implementation is.

The official website (‘About Ruby’, n.d.) states that the reference implementation is CRuby, which itself changes the implementation on every version upgrade, as the language is always being developed. There is also an international standard (International Organization for Standardization, 2012) based on compatible features between CRuby 1.8 (released in 2003) and CRuby 1.9 (released in 2010), but an interpreter implementing this obsolete definition of Ruby language will likely not be able to run modern software, which was written with, for example, CRuby 3.2 (released in 2022) implementation in mind.

The ambiguity is getting worse as developers of some interpreters (goruby, 2014/2022; Lin, 2015/2022) directly state in their READMEs that some language features of the CRuby implementation are not supported — mostly because it is a work in progress done by enthusiasts.

This, again, means that we can not expect all the interpreters to implement the same version of the Ruby language. However, we were able to set some criteria which the interpreters suitable for our research should correspond to.

2.2. CRITERIA TO CHOOSE THE INTERPRETERS

As (in chapter 1) we stated that the benchmark application should behave the same way as the workload the interpreter would encounter in a typical Rails application. So, the main criterion for choosing the benchmark application was its correspondence with industry's practical needs.

We consider it suitable to use the same criterion for choosing the interpreters, which means that if an interpreter runs the benchmark application correctly (that is without errors and with the expected result), then this interpreter suits for the needs of our research. This means that we can choose the suitable interpreters by a simple experiment: start the benchmark application server using the interpreter being tested, request the endpoint we are going to benchmark during the actual experiment, and if the endpoint works as expected and do not throw any errors, it means that the interpreter is suitable for our research.

Interpreters also may have different built-in optimisations like JIT, which may behave unexpectedly and slow down the interpreter in some applications (Kokubun, 2021) and therefore such interpreters provide flags to disable these optimisations. We consider it suitable to measure performance of such interpreters with all possible states of these optimisation flags.

We also consider it suitable to compare performance between actual major versions of CRuby, that is CRuby 2 and CRuby 3, as there is a "Ruby 3x3" initiative to speed up the official Ruby interpreter three times between these two major versions (Matsumoto et al., 2016). The latest available CRuby versions at the moment of writing are 2.7.8 and 3.2.2 ('Ruby Releases', n.d.).

We will also exclude interpreters which are not compatible with x86_64 architecture (though we doubt there are any, as it is the most popular architecture for both servers and workstations), as measuring the performance on systems with other architectures is out of scope of our research.

To ease reproduction of the results, we also limited the interpreters to the ones available in ruby-build.

2.3. SELECTED INTERPRETERS

Each of the selected interpreters is represented as a separate subchapter. The name of a subchapter is the version tag used by ruby-build to install this interpreter of Ruby. The subchapters contain a brief overview of the interpreter and its possible optimisation options.

2.3.1. 2.7.8

The official CRuby interpreter, version 2.7.8. Has a built-in MJIT optimisation feature, but it is still considered experimental and turned off by default. As stated by authors, 2.7's implementation of JIT does not meet the goal of optimising Rails applications (Kokubun, 2019, 2021).

In our experiment the performance was measured with disabled and enabled MJIT.

2.3.2. 3.2.2

The official CRuby interpreter, version 3.2.2. Has built-in MJIT (experimental) and YJIT (built by Shopify and used in their production systems) optimisation features. The latter one is claimed to seriously improve performance of Rails applications, which is interesting to measure.

In our experiment the performance was measured with disabled JIT, enabled MJIT and enabled YJIT.

2.3.3. jruby-9.4.2.0

JRuby is an implementation of the Ruby language using the JVM. "It aims to be a complete, correct and fast implementation of Ruby, at the same time as providing powerful new features such as concurrency without a global-interpreter-lock, true parallelism, and tight integration to the Java language to allow you to use Java classes in your Ruby program and to allow JRuby to be embedded into a Java application" (jruby, 2009/2023).

In our experiment the performance was measured with enabled (default) and disabled ("--dev" option) optimisations.

2.3.4. truffleruby+graalvm-22.3.1

TruffleRuby is a high-performance implementation of the Ruby programming language based on GraalVM. Aims to implement CRuby 3.1 version of the language, but it is not 100% compatible yet (truffleruby, 2016/2023). does not have GIL. Can be run on GraalVM (Native configuration) and on JVM, but it uses a different approach than JRuby, which means it may demonstrate different performance from both JRuby and its own Native configuration.

In our experiment the performance of both Native (default) and JVM configurations was measured.

2.4. NON-SELECTED INTERPRETERS

Each of the non-selected interpreters is represented as a separate subchapter. The name of a subchapter is the version tag used by ruby-build to install this interpreter of Ruby. The subchapters contain a brief overview of the interpreter and reasons why it has not been selected for the research.

2.4.1. artichoke-dev

Artichoke is a Ruby implementation written in Rust and Ruby. Artichoke intends to be CRuby-compatible and currently targets CRuby 3.1.2 (artichoke, 2019/2023).

It does not have the gem manager, so Rails installation was impossible.

2.4.2. maglev-2.0.0-dev

MagLev is an open source implementation of the Ruby programming language and libraries built on top of VMware's GemStone/S 3.1 Virtual Machine. Interestingly, MagLev also has database-like capabilities: it is able to store objects across program instances in a special "Maglev::PERSISTENT_ROOT" variable and claims to have "fully ACID transactions, and

enterprise class NoSQL data management capabilities to provide a robust and durable programming platform" ('MagLev', n.d.).

It is past its end of life and is now unsupported. Implements a very old version of Ruby (CRuby 1.9). Latest updates in the repository seem to be around 7 years ago.

2.4.3. mruby-3.2.0

Mruby is a lightweight Ruby interpreter, which implements the ISO standard (CRuby 1.8 – 1.9) with more recent features provided by CRuby 3. It is designed to be embedded into other applications.

It has its own gem manager, but not the standard one, so Rails installation was impossible.

2.4.4. picoruby-3.0.0

Picoruby is an alternative mruby implementation designed for microcontrollers.

Since it does not have the gem manager, the Rails installation was impossible.

2.4.5. rbx-5.0

Rubinius is a Ruby interpreter and virtual machine written in Ruby. It aims to be compatible with the most recent stable Ruby version.

It is past its end of life and is now unsupported. Seems to be implementing a very old version of Ruby. Latest updates in the repository seem to be around 3 years ago.

2.4.6. ree-1.8.7-2012.02

Ruby Enterprise Edition is a commercial version of Ruby designed to be used with nginx and Phusion Passenger.

It is past its end of life and is now unsupported, as its developers wished to focus their efforts on Phusion Passenger and other products many more than 10 years ago (Lai, 2012). Implements a very old version of Ruby (1.8.7).

2.4.7. truffleruby-22.3.1

This version duplicates truffleruby+graalvm-22.3.1, but is distributed as a standalone binary with embedded GraalVM and allows only Native configuration.

3. CONDUCTING THE EXPERIMENT

3.1. EXPERIMENT DESIGN

The virtual server which ruan the benchmark application was an AWS m4.2xlarge instance (8 vCPUs) which ran on Ubuntu Server 22.04 LTS (AMI ID: ami-0a695f0d95cefc163). The server ran railsbench based on Rails 6.1.6.1 served by puma under each benchmarked interpreter. The application startup time was also measured. Puma was set to have 16 threads (2 per each vCPU) and one worker process (to accommodate the fact that JVM does not support multiple processes).

There were two other virtual servers in the same local network. One of them ran PostgreSQL server, one of them ran the benchmark tool. The benchmark tool performs 65536 requests to the endpoint being tested during the benchmark in batches of 1024 (64 batches total), up to 32 simultaneously (2 times more than the amount of the processing threads to always keep the server overloaded and simulate peak load this way). Since we aimed to see the interpreter's behaviour during the warmup too, we started the benchmark right after the application's startup.

The experiment was repeated three times for each interpreter to make sure the results are reliable and not influenced by random factors.

All source data related to this experiment and the benchmarking script are published to a GitHub repository (Kazakov, 2023/2023).

3.1.1. Prevention of the ephemeral port exhaustion issue

By default, the benchmarking tool used for the experiment (ab) would open one TCP connection per one HTTP request. It means that it would open tens of thousands TCP connections and close them within a short time interval (tens of seconds) during the

experiment. As per TCP specification (Internet Engineering Task Force, 1981), each socket will have to stay in TIME-WAIT state for 2*MSL interval after it is closed and, therefore, keep the host:port pair busy during that interval (which is usually measured in minutes), which will lead to ephemeral port exhaustion, as was discovered while preparing the experiment.

The possible workarounds are:

1. Reducing the MSL value and increasing the ephemeral ports range, which means tuning the operating system's network stack, which is out of scope of this research
2. Using IP_BIND_ADDRESS_NO_PORT option for outgoing connections (Majkowski, 2022), but the benchmarking tool does not seem to support this
3. Instead of these, we used HTTP keep-alive feature, which is supported by the benchmarking tool and is actually closer to a real-world scenario, where incoming connections are forwarded to microservices by a load balancer.

3.1.2. Modifications to the benchmark application

1. Removed gems unnecessary for the measurements from railsbench's Gemfile: sass-rails, webpacker (as we did not need to compile assets), jbuilder (this does not seem to be used in the app at all and looks like a residue from default rails application generator) and also all gems from "group: :development" blocks (as we ran the application only with RAILS_ENV=production)
2. Added a pg gem to make the app work with PostgreSQL. While measuring using JRuby interpreter, we replaced it with activerecord-jdbcpostgresql-adapter gem, as the pg gem is not compatible with JRuby
3. Added a "respond_to" block to the PostsController#show endpoint to make it render the post as JSON (it returned HTTP 406 Not Acceptable error without it)
4. Modified the config/puma.rb file to set maximum and minimum thread count to 16
5. Modified the config/database.yml file to add PostgreSQL host, user and password and set the database connection pool size to 32 (though only 16 connections to the PostgreSQL server were actually initiated during the experiment due to the limitation of 16 puma threads).

3.2. EXPERIMENT RESULTS

This chapter shows the experiment data obtained for each of the interpreters with various optimisation flags.

The name of each subchapter is the version tag used by ruby-build to install this interpreter of Ruby, followed by the optimisation flag used in this experiment. Each subchapter contains an output of “ruby --version” command for the interpreter, a table with application load time and RPS values (average between all batches and for the last batch, that is after warm up) for each measurement and figure with a p50 and p99 request times comparison charts between measurements (batch number on X-axis, request durations on Y-axis).

Explanation and analysis of the results are given in chapter 3.3.

3.2.1. 2.7.8

Version line: ruby 2.7.8p225 (2023-03-30 revision 1f4d455848) [x86_64-linux]

Table 1. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	1.17	1.16	1.18	1.17
Average requests per second	374.31	375.29	377.67	375.76
Last batch requests per second	373.43	377.34	376.29	375.69

Data source: the present measurements. Composed by the author.

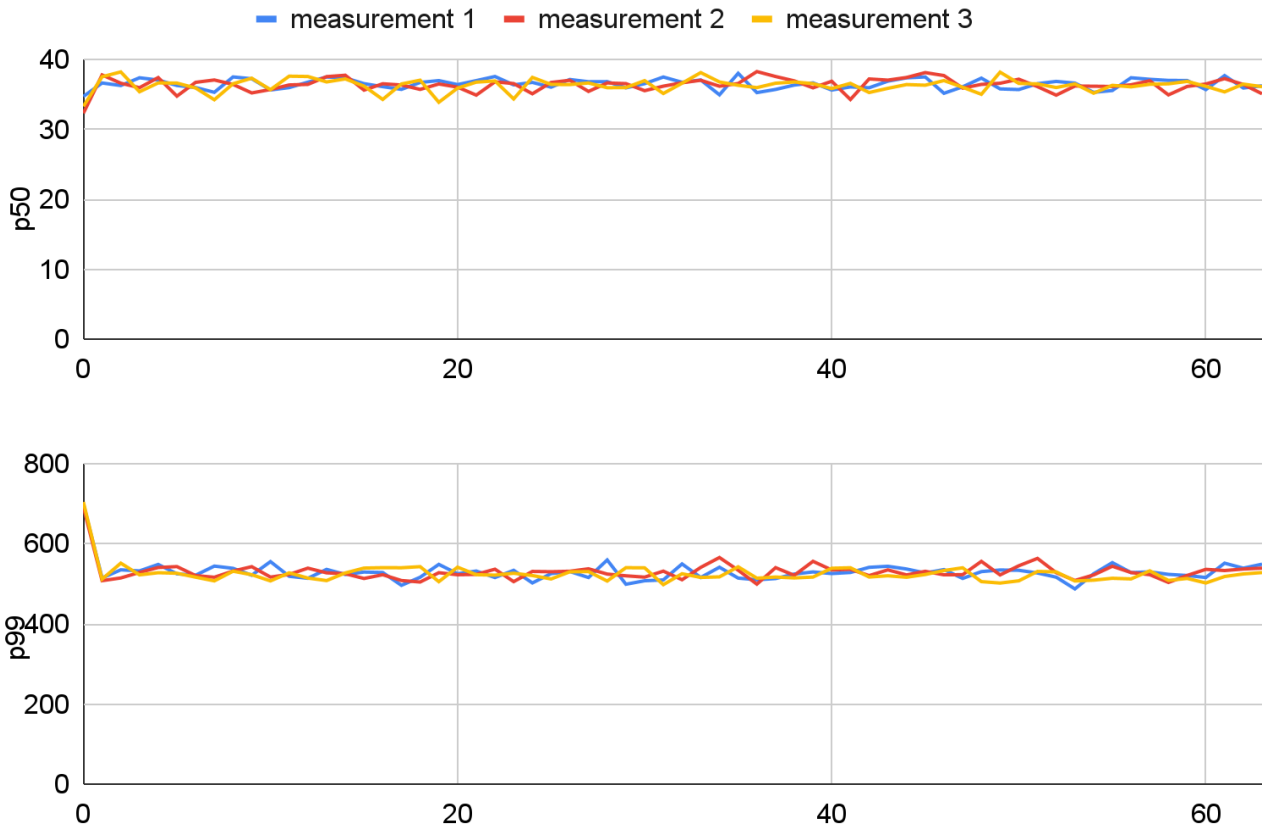

Figure 1. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.2.2. 2.7.8 --jit

Version line: ruby 2.7.8p225 (2023-03-30 revision 1f4d455848) +JIT [x86_64-linux]

Table 2. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	1.23	1.23	1.23	1.23
Average requests per second	356.78	358.91	358.51	358.07
Last batch requests per second	360.16	362.29	372.24	364.90

Data source: the present measurements. Composed by the author.

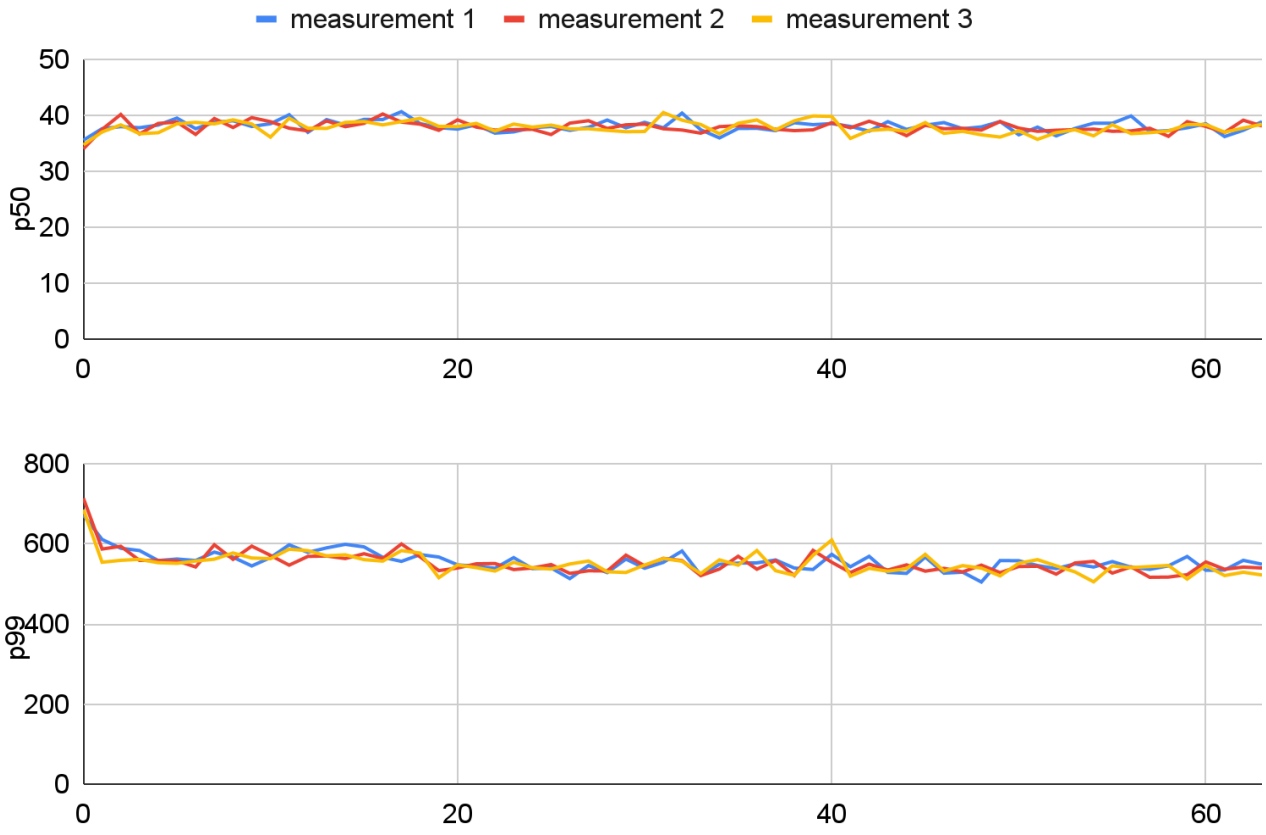

Figure 2. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.2.3. 3.2.2

Version line: ruby 3.2.2 (2023-03-30 revision e51014f9c0) [x86_64-linux]

Table 3. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	1.15	1.15	1.15	1.15
Average requests per second	395.24	393.45	395.12	394.60
Last batch requests per second	394.75	393.63	393.53	393.97

Data source: the present measurements. Composed by the author.

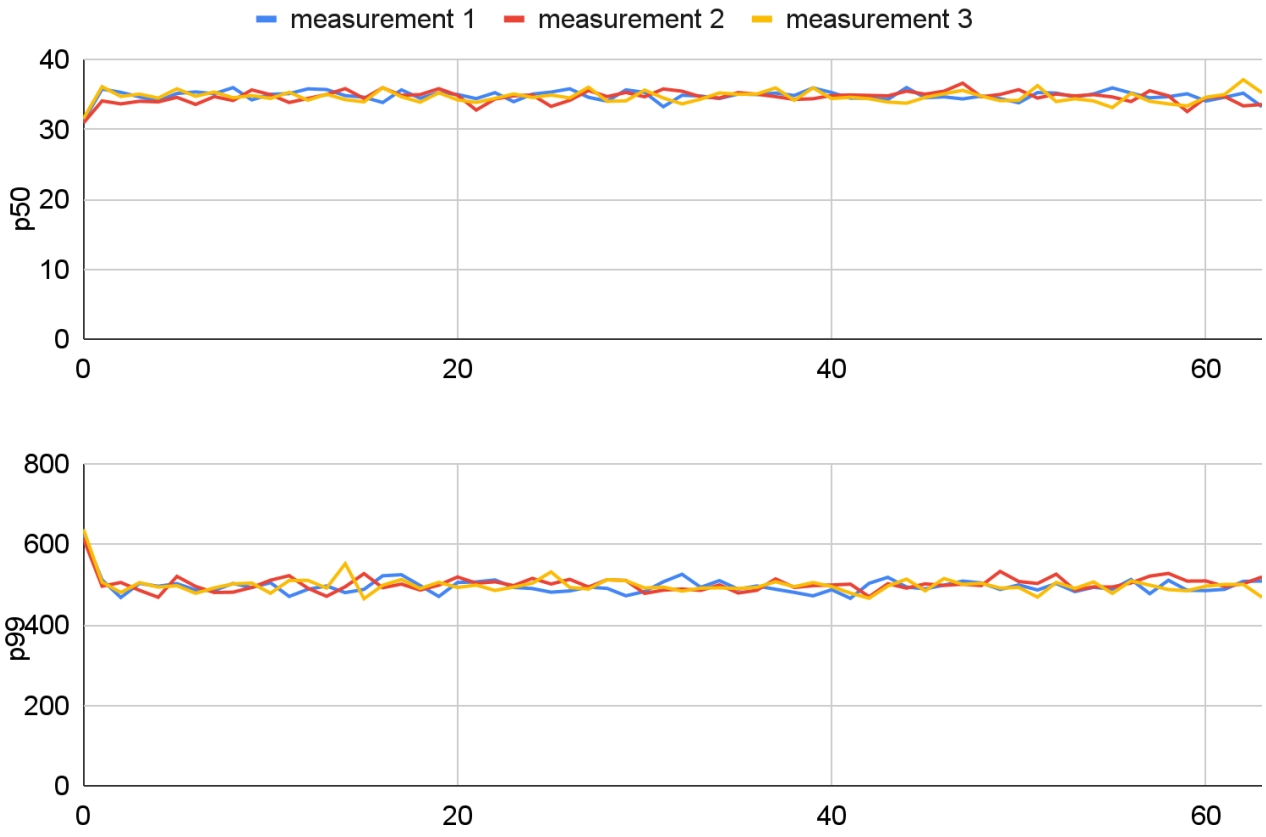

Figure 3. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.2.4. 3.2.2 --mjit

Version line: ruby 3.2.2 (2023-03-30 revision e51014f9c0) +MJIT [x86_64-linux]

Table 4. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	1.89	1.89	1.91	1.90
Average requests per second	396.88	375.66	378.15	383.56
Last batch requests per second	397.38	393.35	396.80	395.84

Data source: the present measurements. Composed by the author.

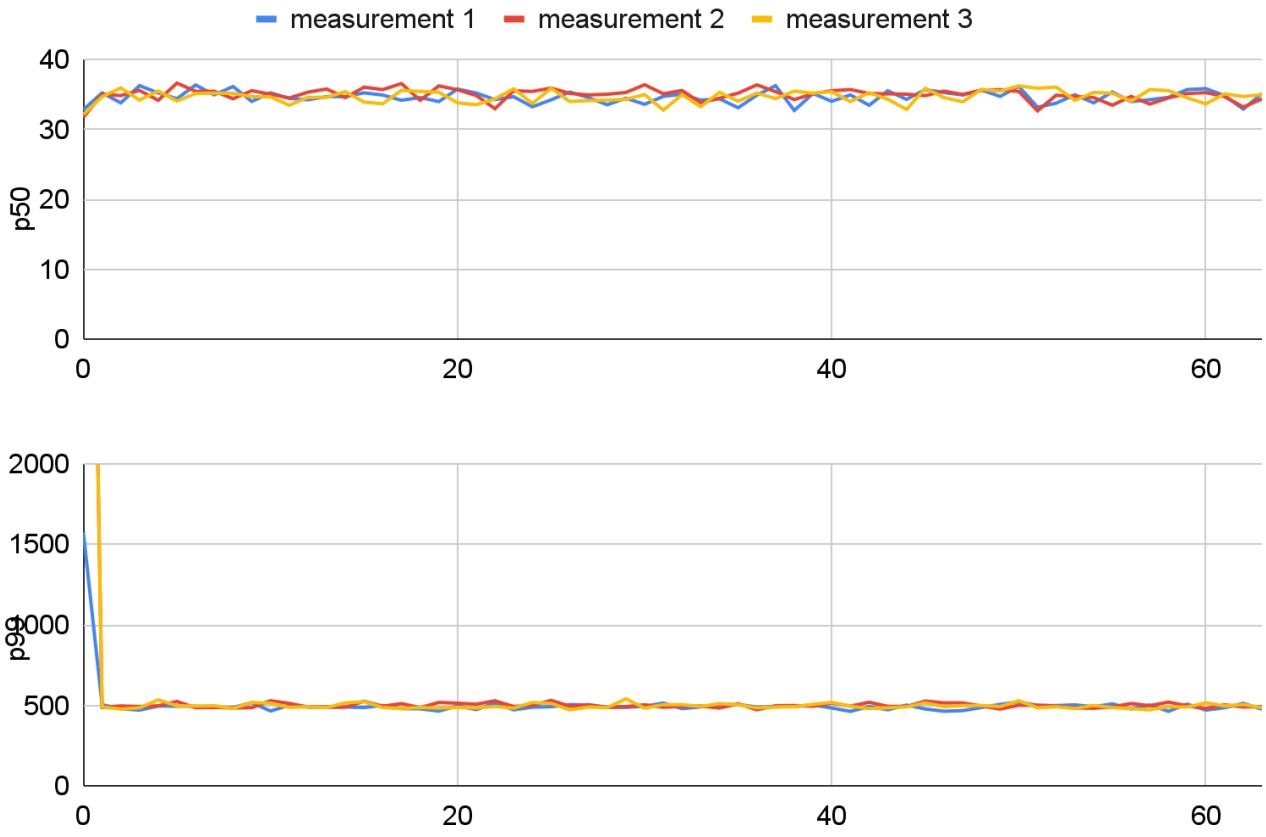

Figure 4. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.4.5. 3.2.2 --yjit

Version line: ruby 3.2.2 (2023-03-30 revision e51014f9c0) +YJIT [x86_64-linux]

Table 5. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	1.40	1.40	1.37	1.39
Average requests per second	482.68	487.61	487.82	486.04
Last batch requests per second	491.63	484.38	495.90	490.64

Data source: the present measurements. Composed by the author.

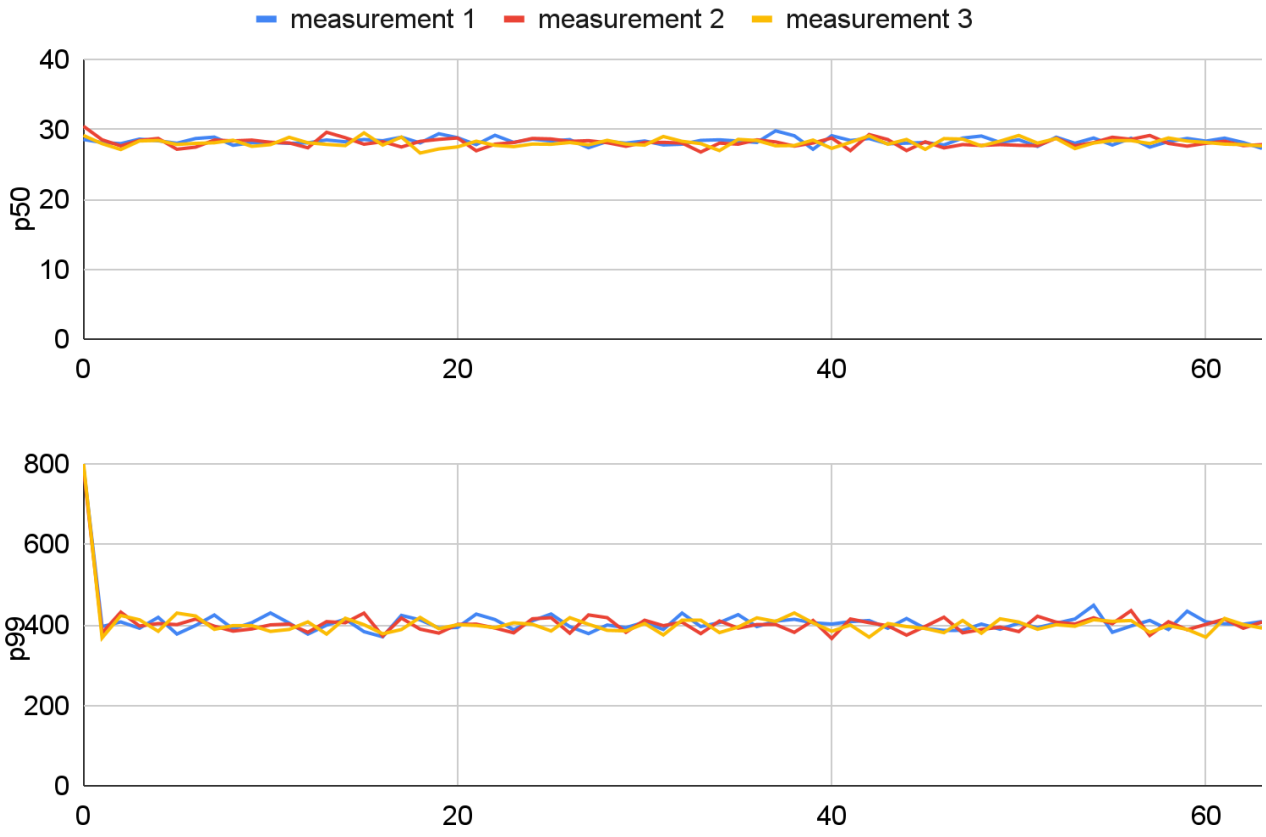

Figure 5. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.4.6. jruby-9.4.2.0

Version line: jruby 9.4.2.0 (3.1.0) 2023-03-08 90d2913fda OpenJDK 64-Bit Server VM 19.0.2+7-Ubuntu-0ubuntu322.04 on 19.0.2+7-Ubuntu-0ubuntu322.04 +jit [x86_64-linux]

Table 6. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	9.01	8.97	9.32	9.10
Average requests per second	660.80	655.94	654.29	657.01
Last batch requests per second	726.90	734.23	725.91	729.01

Data source: the present measurements. Composed by the author.

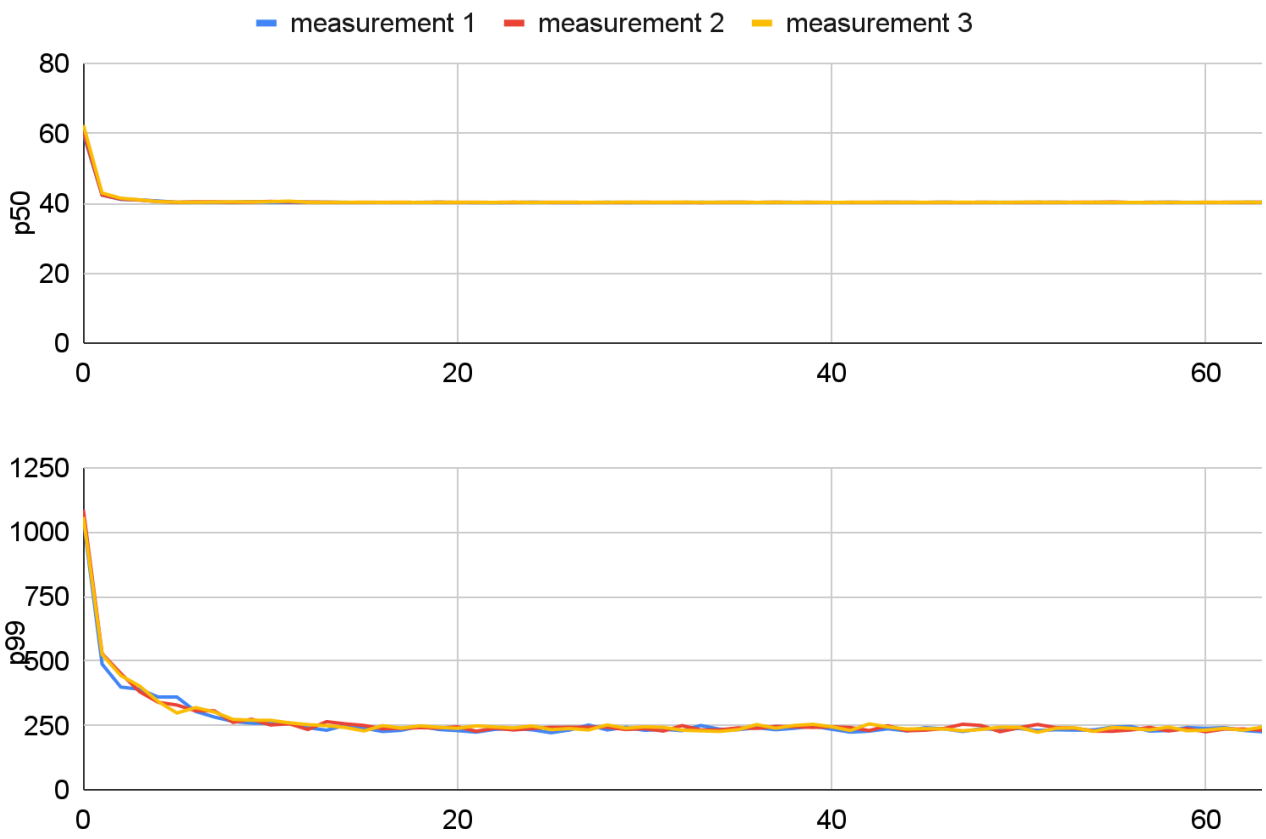

Figure 6. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.4.7. jruby-9.4.2.0 --dev

Version line: jruby 9.4.2.0 (3.1.0) 2023-03-08 90d2913fda OpenJDK 64-Bit Server VM 19.0.2+7-Ubuntu-0ubuntu322.04 on 19.0.2+7-Ubuntu-0ubuntu322.04 [x86_64-linux]

Table 7. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	7.31	7.24	6.98	7.18
Average requests per second	511.71	511.40	508.29	510.47
Last batch requests per second	513.48	514.85	511.20	513.18

Data source: the present measurements. Composed by the author.

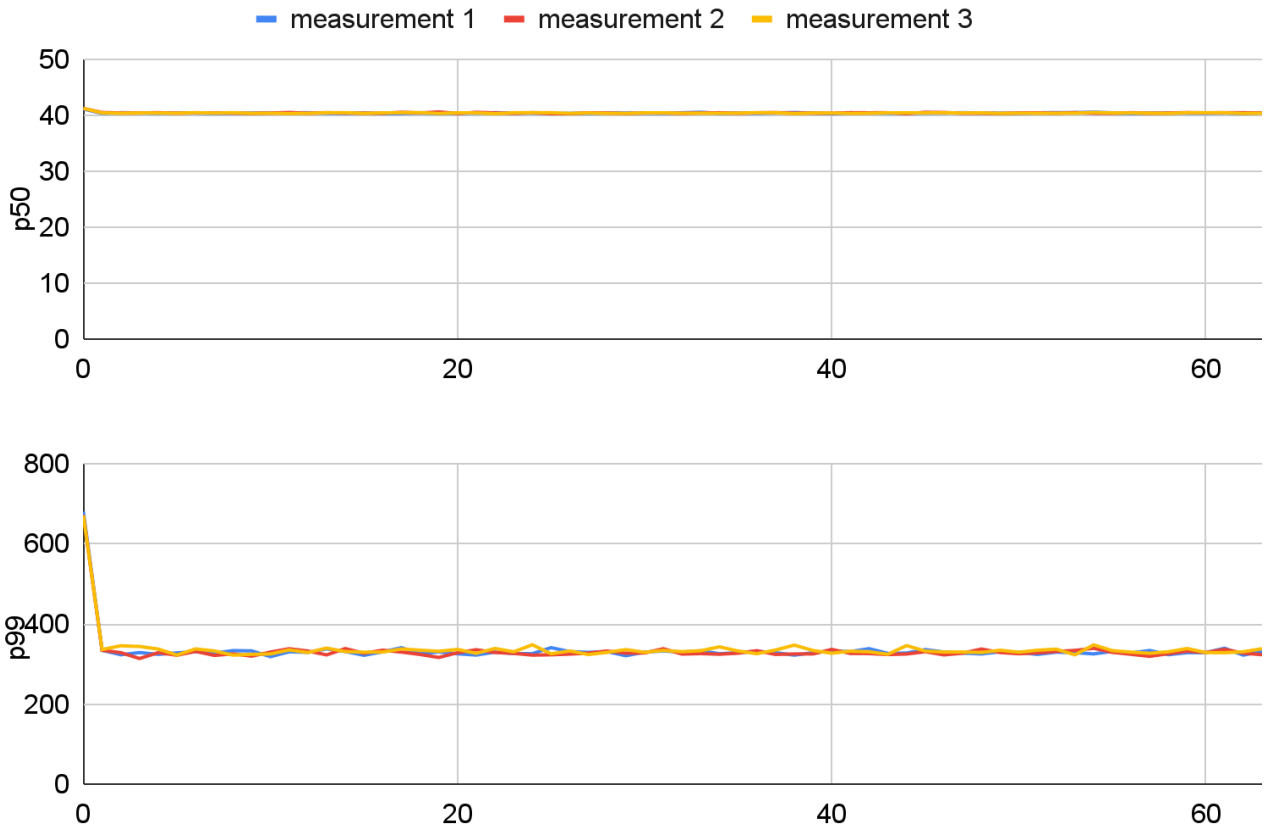

Figure 7. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.4.8. truffleruby+graalvm-22.3.1

Version line: truffleruby 22.3.1, like ruby 3.0.3, GraalVM CE Native [x86_64-linux]

Table 8. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	12.24	12.01	12.20	12.15
Average requests per second	257.85	193.43	199.74	217.01
Last batch requests per second	374.23	127.66	135.37	212.42

Data source: the present measurements. Composed by the author.

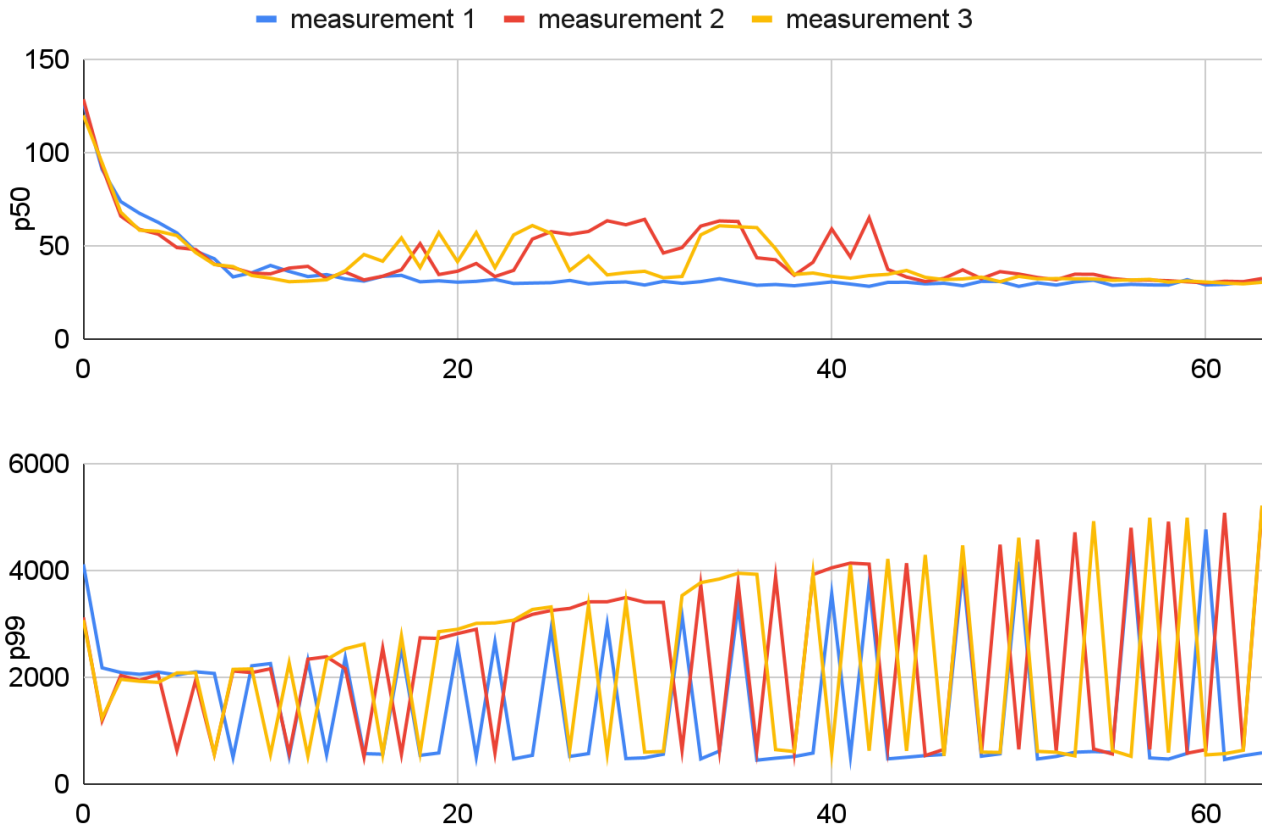

Figure 8. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.4.9. truffleruby+graalvm-22.3.1 --jvm

Version line: truffleruby 22.3.1, like ruby 3.0.3, GraalVM CE JVM [x86_64-linux]

Table 9. Measured application load time and RPS for this interpreter

Measurement	**1**	**2**	**3**	**Average**
Application load time, seconds	20.82	22.00	22.90	21.91
Average requests per second	462.83	454.29	465.87	416.00
Last batch requests per second	591.71	601.99	592.99	595.56

Data source: the present measurements. Composed by the author.

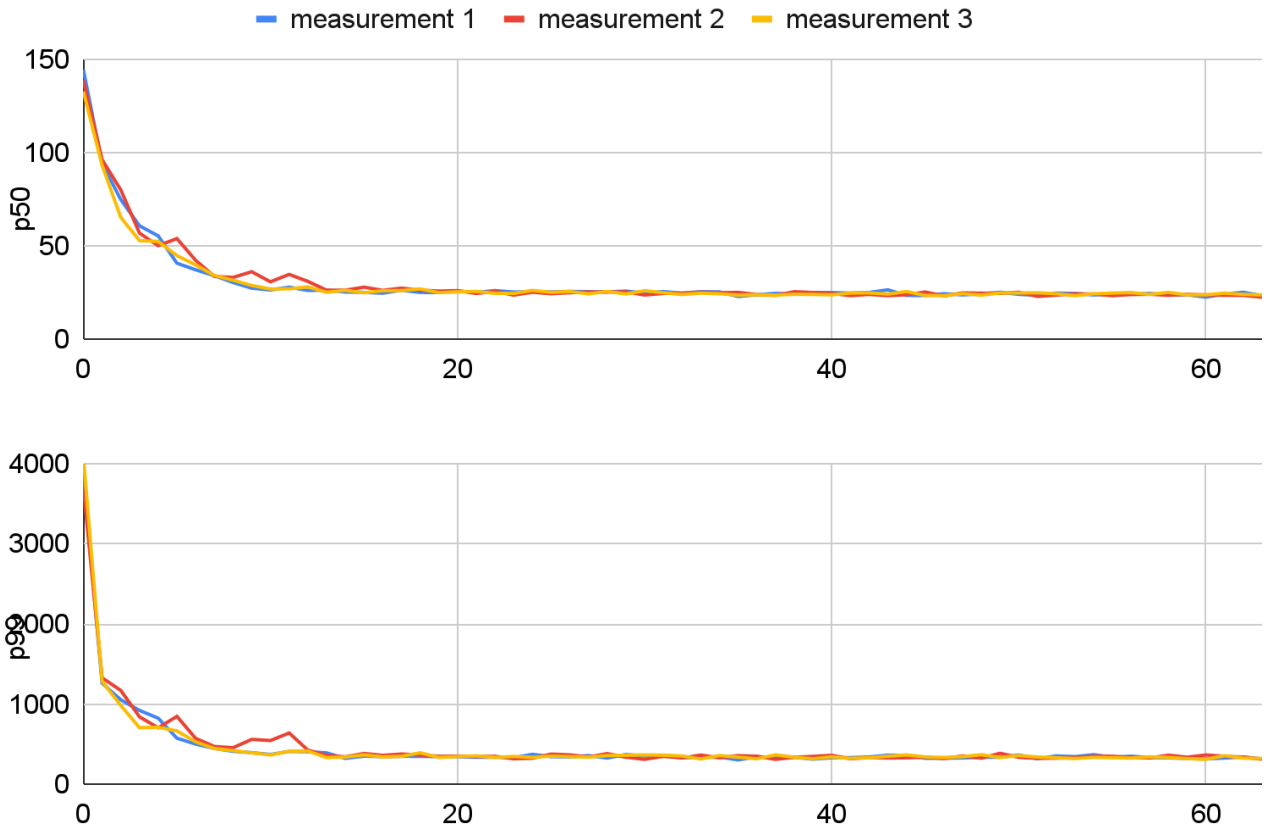

Figure 9. Comparison of p50 and p99 response times between all three measurements. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

3.3. PERFORMANCE COMPARISON

3.3.1. truffleruby+graalvm-22.3.1 anomaly

The truffleruby+graalvm-22.3.1 (default, Native configuration) was excluded from the comparison, as the response time charts (as well as RPS measurements) demonstrated a very high discrepancy between all three measurements.

The identification of underlying reasons was out of scope of this research, however, the reference manual (‘Reporting Performance Problems’, n.d.) allows to assume the following possible reasons:

- The manual states that “to experiment with how fast TruffleRuby can be, we recommend using the Enterprise Edition of GraalVM”. Within our design we installed the community version from ruby-build (as it is a limitation of our research), which, according to the official download page (‘Download GraalVM’, n.d.), lacks “patented advanced compiler optimizations”, “compressed pointers for low memory usage”, “profile guided optimization for improved performance” and “G1 garbage collection for low latency”, which are, indeed, performance optimisation features that could have resulted in more steady outcome
- It also recommends to run with the --engine.TraceCompilation flag to check for basic performance problems, as some methods during the JIT compilation process can fail to compile or be mistakenly compiled several times, and states that “if you do not run with this flag, TruffleRuby will try to work around errors and you will not see that there is a problem”. In-depth investigation was out of scope of our research, but when we were preparing the experiment on a local machine, we repeatedly saw TruffleRuby crashes with segmentation fault error. Our assumption was that during the measurements it was also failing, demonstrating spikes on p50 and p99 charts instead of crashing.

However, TruffleRuby with JVM configuration does not demonstrate such discrepancy, so it was kept in the comparison.

3.3.2. Overall comparison

The figures 10 and 11 show p50, p99 and RPS values for all the interpreters.

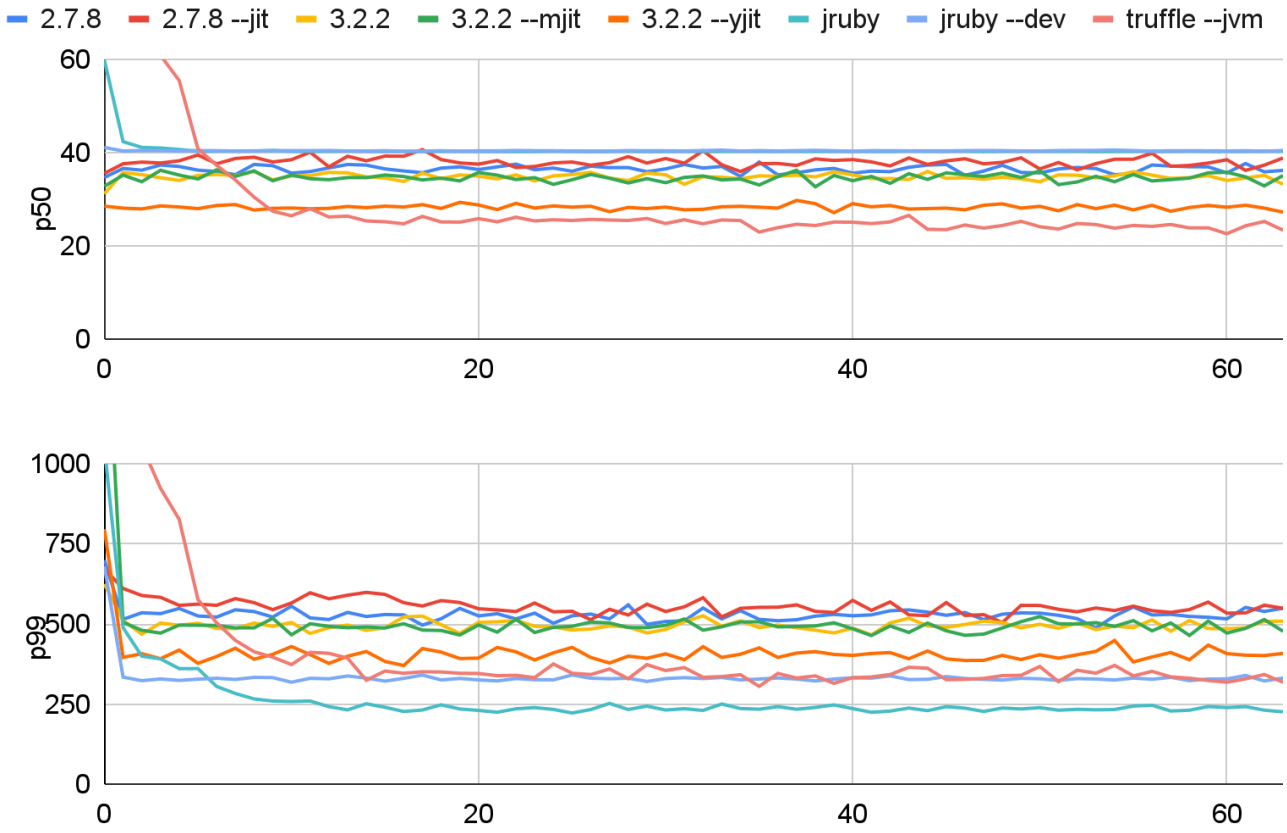

Figure 10. Comparison of p50 and p99 response times between interpreters. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

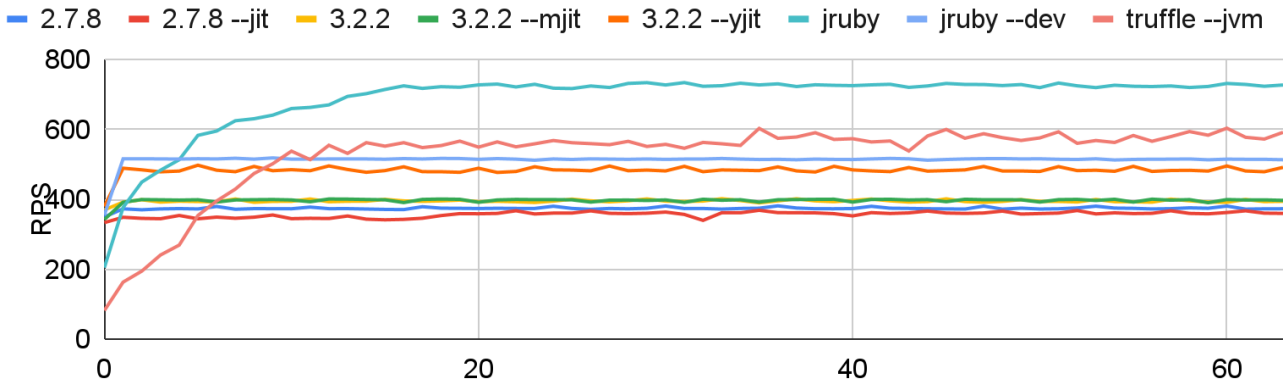

Figure 11. RPS comparison between interpreters. Y-axis: RPS, X-axis: batch number. Measured and composed by the author

These charts demonstrate some important trends:

- All CRuby interpreters seem to be within the same range of performance, excluding 3.2.2 with YJIT enabled, which behaves more like JRuby with disabled optimisations
- JIT-enabled interpreters (especially JVM-based ones) need some time to warm up until they get to their peak performance, which can be seen as very characteristic L-shaped (or Γ-shaped in case of RPS) lines
- All interpreters seem to have been warmed up after about 16-th batch (16384 requests), as the chart lines stabilise after this batch
- At the same time, CRuby JIT-enabled interpreters seem to have been already warmed up after the first batch (1024 requests)
- After warm-up, JRuby in default configuration is the obvious leader in terms of p99 response time, which also makes it a leader in terms of RPS
- At the same time, TruffleRuby seems to be ahead of it on the p50 chart
- p99 values are a magnitude of order higher that p50 ones (if we look at the source data, we can notice the same between minimal and p50 values)
- JRuby's chart lines look much smoother than other interpreters, so for some reason it demonstrates more stable performance between batches.

Looking at the load times, we can additionally see the "more optimisations — longer load time" trend:

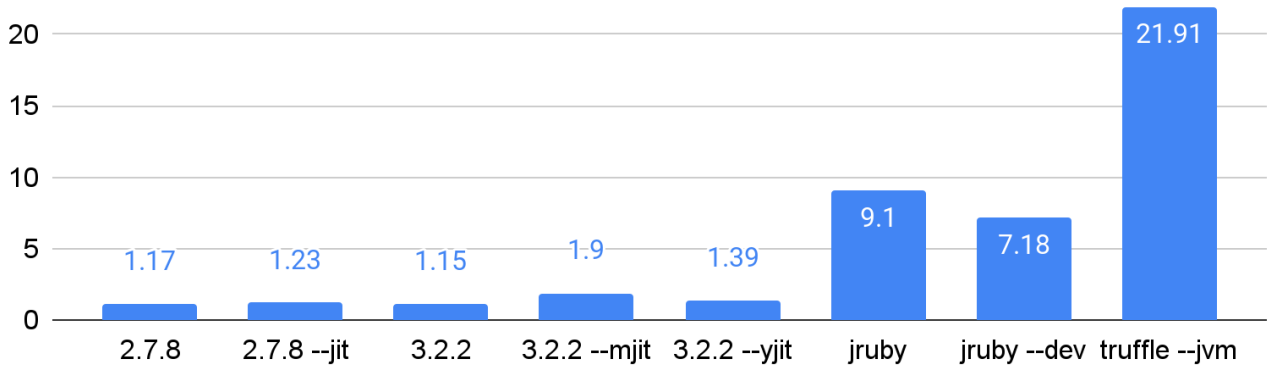

Figure 12. Load time comparison between interpreters. Y-axis: average load time in ms, X-axis: interpreter name and optimisation flags. Measured and composed by the author

3.3.3. CRuby interpreters

The next figures show p50, p99 and RPS values for CRuby interpreters.

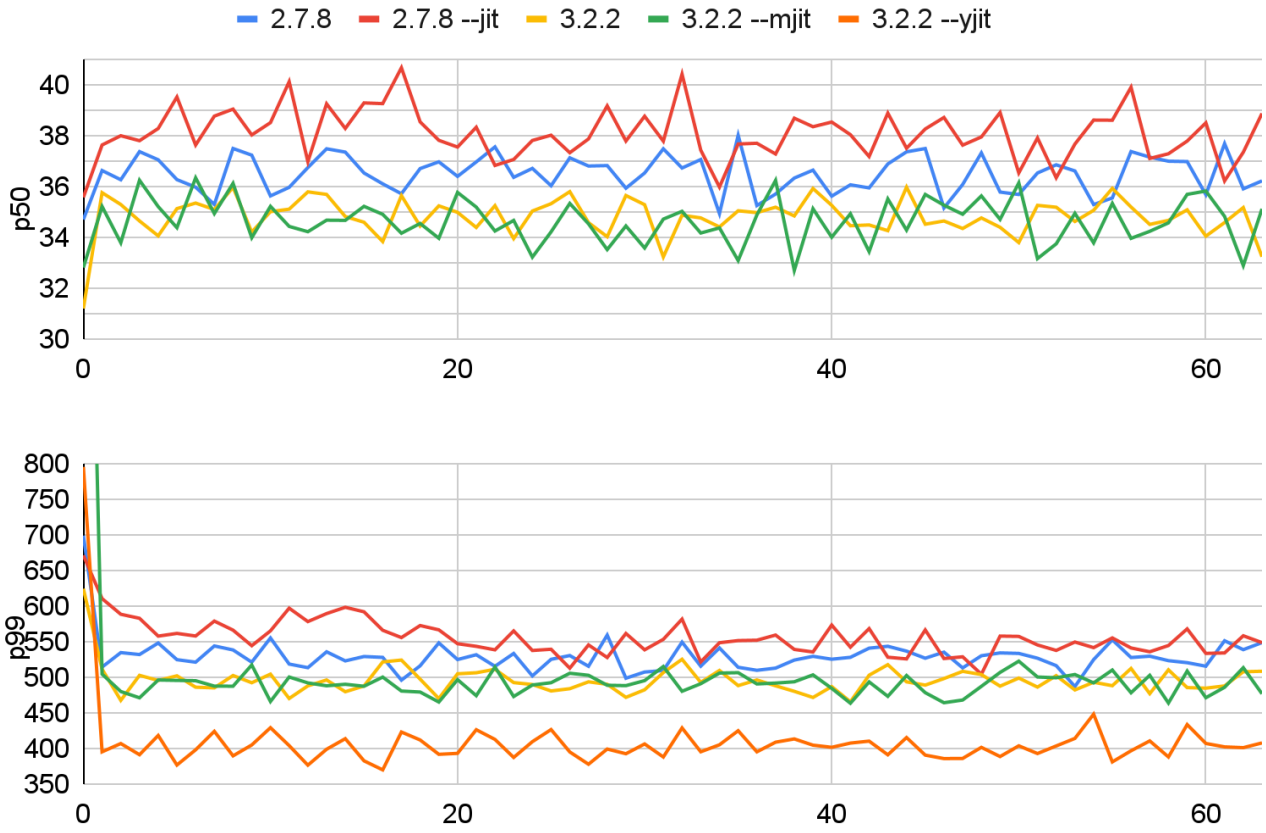

Figure 13. Comparison of p50 and p99 response times between interpreters. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

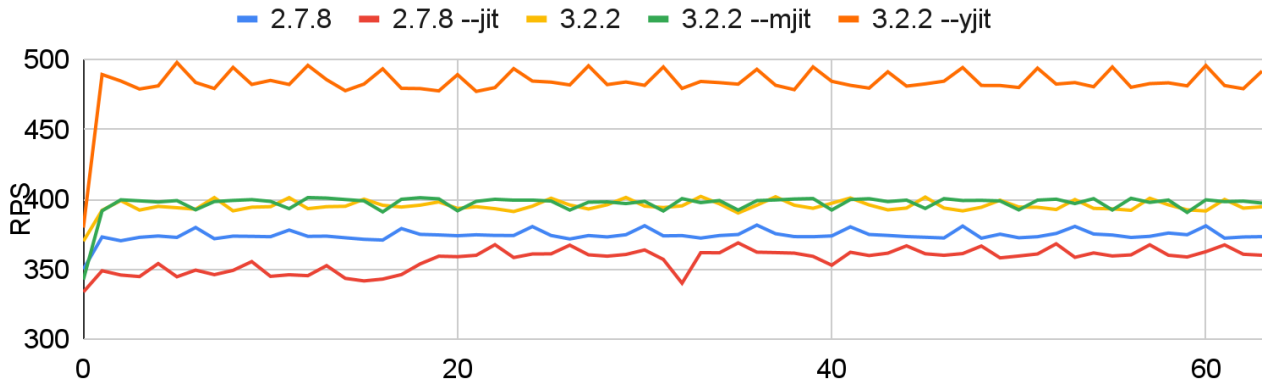

Figure 14. RPS comparison between interpreters. Y-axis: RPS, X-axis: batch number. Measured and composed by the author

First of all, these measurements confirm Kokubun's thought which we referenced in chapters 2.2 and 2.3.1, that sometimes optimisations can be inefficient in certain applications, and Rails is definitely one of these applications:

- 2.7.8's JIT implementation actually slows it down which is demonstrated on the RPS chart. The result seems to be improving between batches 16 and 20, but still stays behind 2.7.8 without JIT
- 3.2.2's MJIT implementation shows less than 0,5% positive difference in last batch's RPS, which can not be considered a reliable improvement against 3.2.2 without JIT.

However, 3.2.2's YJIT implementation shows a significant 25% positive difference against 3.2.2 without JIT. It also shows that the warmup process is extremely fast: it takes just one batch of 1024 requests. Presumably, both of these observations are related to the fact that YJIT compiles all methods that are called more than once. "If you compile all benchmarked methods instead of just the top 100 methods, the JIT makes Rails faster" (Kokubun, 2021).

Regarding the difference between 2.7.8 and 3.2.2 without JIT, 3.2.2 shows 5% better performance in Rails than 2.7.8.

3.3.4. JVM-based interpreters

The following figures show the p50, p99 and RPS values for JVM-based interpreters.

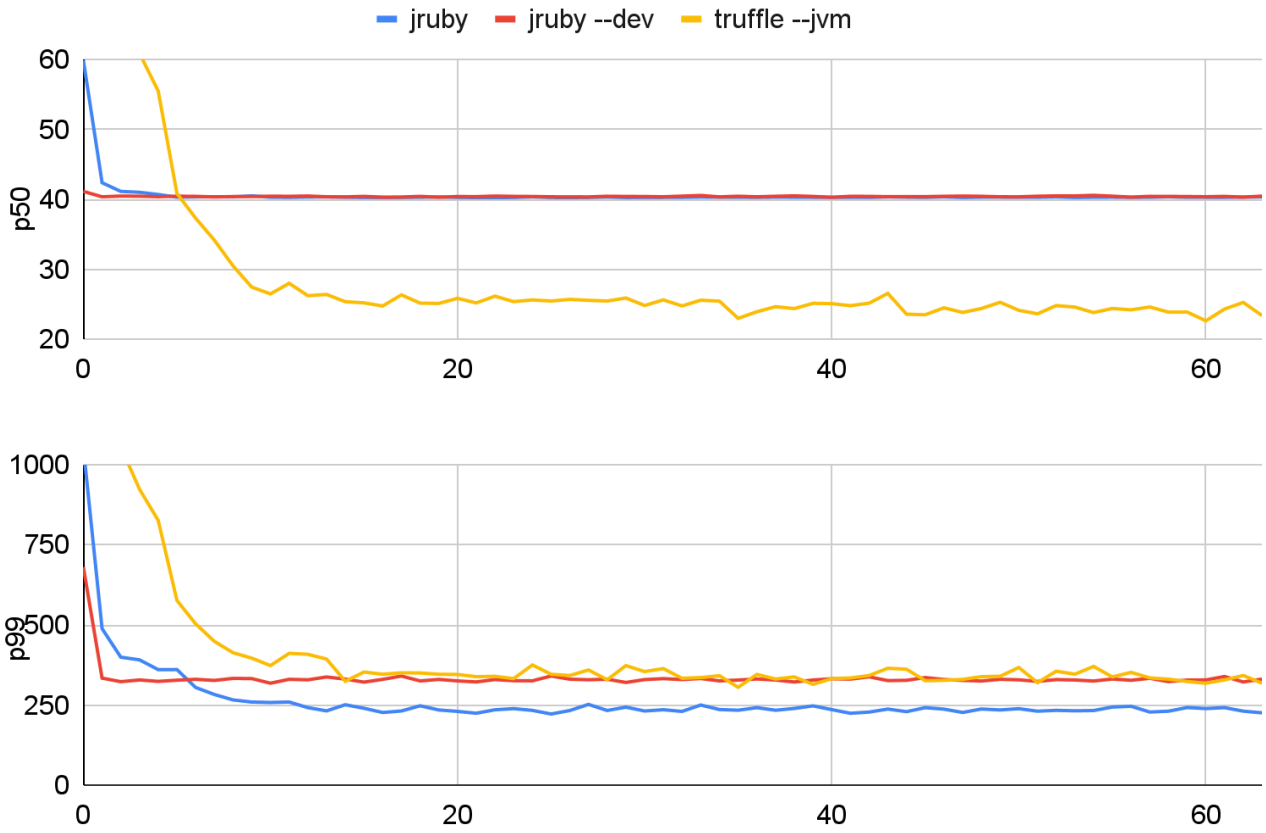

Figure 15. Comparison of p50 and p99 response times between interpreters. Y-axis: response times in ms, X-axis: batch number. Measured and composed by the author

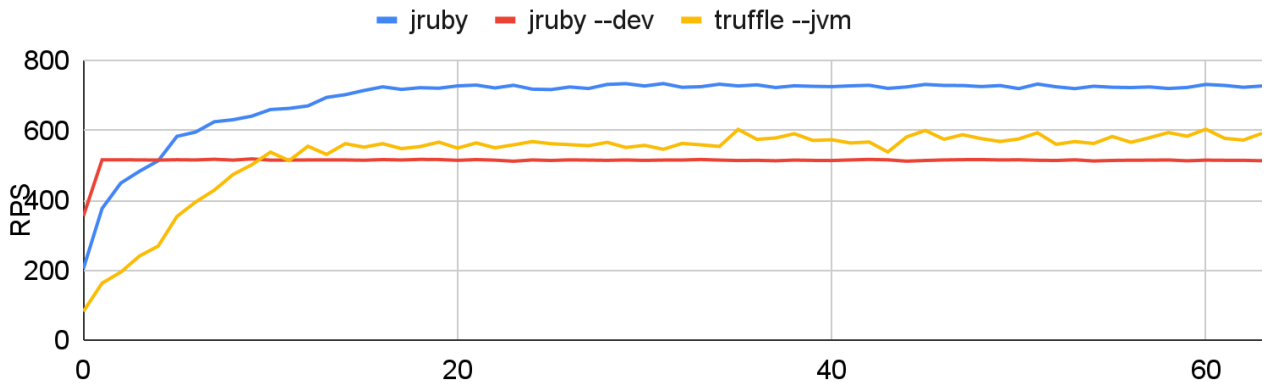

Figure 16. RPS comparison between interpreters. Y-axis: RPS, X-axis: batch number. Measured and composed by the author

JVM-based interpreters demonstrate very interesting results. They all outperform even the fastest CRuby (3.2.2 --yjit) in terms of RPS (at the expense of application start time, which is many times higher than in case of CRuby). There are few possible explanations:

- Even with most of optimisations optimisations disabled by --dev flag, JVM has better GC and memory management strategies
- JVM has true parallelism between threads and does not have GIL, which can also explain less difference between p50 and p99 values (as requests have to wait less in queues) and more stable results between batches.

It is also interesting that JRuby optimises the p99 value (71% improvement for JRuby's p99 against TruffleRuby's; equal to JRuby with JIT vs JRuby without JIT), while TruffleRuby does it for p50 (72% improvement for TruffleRuby's p50 against JRuby's).

Such improvements happen at the cost of increased load time and likely increased memory footprint, but measuring the memory footprint is out of scope of our research.

3.3.5. Last batch RPS comparison

The figure 17 shows the last batch (after the warmup) RPS for every interpreter.

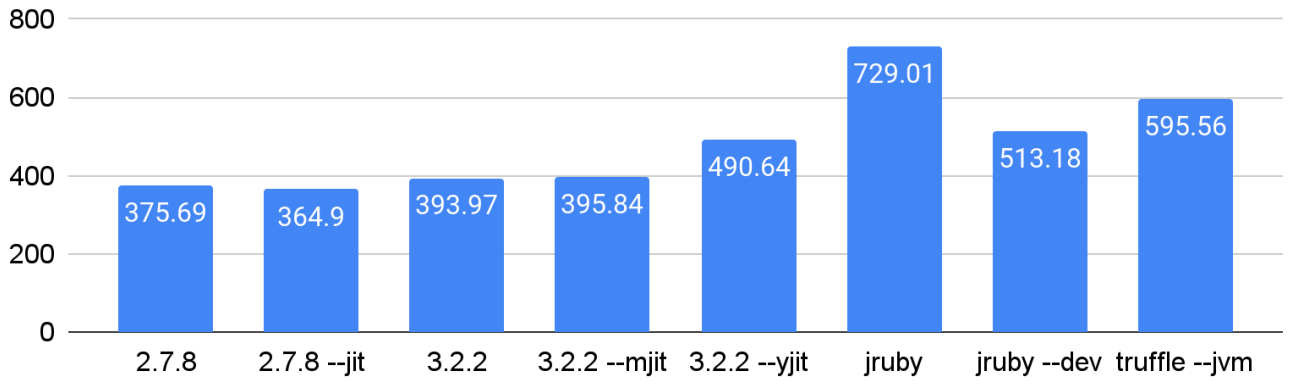

Figure 17. Last batch RPS comparison between interpreters. Y-axis: average last batch RPS, X-axis: interpreter name. Measured and composed by the author

3.3.6. Answer to the research question

Alternative Ruby interpreters can show increased (94% compared to CRuby 2.7.8) performance when used in a high-load web backend.

JRuby seems to be the best choice in this regard. If there is no possibility to use JRuby, it is possible to use CRuby 3.2.2 with YJIT enabled, which gives 30% better performance when compared to CRuby 2.7.8 and 25% better performance when compared to CRuby 3.2.2 without JIT.

It is worth noting that any improvements come with a tradeoff. The tradeoffs we clearly observed are:

- The interpreters which show more RPS tend to start up more slowly (though slowest start up does not necessarily mean highest RPS, like in TruffleRuby case)
- The most efficient JIT-enabled interpreters need a significant warmup period (about 16 thousands requests) before plateauing at their peak performance. At startup, they are actually the slowest.

Besides this, it is possible that the fastest interpreters demonstrate other tradeoffs, such as higher memory consumption, CPU load or IO operations amount, but measuring this is out of scope of our research. In our opinion, it does not actually make sense to measure these in such a comparison, as different applications will demonstrate different results. As we described in the “LIMITATIONS” chapter, replacing the interpreter in an existing IT system will require experimenting with the system itself (on the system’s applications and in the system’s environment), including measuring all these values and then making an informed decision based on these facts.

This research may be useful for companies and engineers to set a direction for further experimentation. Our results show that such experimentation indeed makes sense, may help to reduce infrastructure costs and therefore increase business efficiency and also decrease environmental footprint.

CONCLUSION

This research is relevant for companies whose IT systems experience high load and are written in Ruby. It addresses the problem of Ruby's low performance and poses the following research question: can using alternative Ruby interpreters show increased performance in such IT systems?

The research consists of three parts:

1. Preparing the benchmark. This part gives an overview on microservice architecture, which is a best-practice approach for building high-load IT systems, formulates functional and non-functional requirements for the benchmark application and chooses the benchmark application across these which were already used in other studies
2. Selecting suitable interpreters. This part discusses the issue that different interpreters implement slightly different versions of the language, sets the criteria for choosing the interpreters and chooses them for the experiment
3. Conducting the experiment. The experiment is designed as three Amazon AWS virtual servers within the same network, one hosting PostgreSQL server, one hosting the benchmark application, one hosting the benchmarking tool. The benchmarking tool sends 64 batches of 1024 requests, measuring each batch individually to also see the warmup dynamics. The experiment is repeated three times for each interpreter to confirm reliability of the data. This part also describes caveats and analyses the experiment's outcome.

Analysis of the experimental data allowed us to demonstrate that JRuby and TruffleRuby outperformed CRuby in the experiment. At the same time, some tradeoffs were observed, such as longer startup time and long warmup period needed to reach the peak performance. We concluded that better-performing interpreters might possibly come with even more tradeoffs,

though deep analysis of these was out of scope of this work, as these were likely to be individual for every IT system.

This research demonstrated clear direction for companies and engineers for further experimentation with their own IT systems and suggests that using an alternative Ruby interpreter might be advantageous, potentially increasing a system's performance and decreasing infrastructure costs.

REFERENCES

About Ruby. (n.d.). Retrieved 8 January 2023, from https://www.ruby-lang.org/en/about/

artichoke. (2023). *Artichoke Ruby* [Rust]. Artichoke Ruby. Retrieved from https://github.com/artichoke/artichoke (Original work published 2019)

Bini, O. (2008, November 26). Expressive power vs performance | Ola Bini: Programming Language Synchronicity. Retrieved 6 February 2023, from https://olabini.se/blog/2008/11/expressive-power-vs-performance/

Cangiano, A. (2008, December 9). The Great Ruby Shootout (December 2008). Retrieved 11 February 2023, from Programming Zen website: https://programmingzen.com/the-great-ruby-shootout-december-2008/

Chen, R., Li, S., & Li, Z. (2017). From Monolith to Microservices: A Dataflow-Driven Approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 466–475. Nanjing: IEEE. doi: 10.1109/APSEC.2017.53

Chevalier-Boisvert, M., Gibbs, N., Boussier, J., Wu, S. X. (Alan), Patterson, A., Newton, K., & Hawthorn, J. (2021). YJIT: A basic block versioning JIT compiler for CRuby. *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 25–32. Chicago IL USA: ACM. doi: 10.1145/3486606.3486781

codicoscepticos. (2023). *List of Ruby implementations*. Retrieved from https://github.com/codicoscepticos/ruby-implementations (Original work published

2012)

discourse. (2023). *Discourse/discourse* [Ruby]. Discourse. Retrieved from https://github.com/discourse/discourse (Original work published 2013)

Download GraalVM. (n.d.). Retrieved 22 April 2023, from https://www.graalvm.org/downloads/

Dymo, A. (2015). *Ruby performance optimization: Why Ruby is slow, and how to fix it*. Dallas, Texas: The Pragmatic Bookshelf.

Egeonu, E. (2022, June 3). A Roundup of the Top 28 Ruby on Rails Companies in 2022. Retrieved 28 January 2023, from DistantJob—Remote Recruitment Agency website: https://distantjob.com/blog/ruby-on-rails-companies/

Fowler, M., & Lewis, J. (2014, March 25). Microservices. Retrieved 8 January 2023, from Martinfowler.com website: https://martinfowler.com/articles/microservices.html

Ganguly, D. S. (2018, April 11). Pros & Cons you must know before using Ruby on Rails for your startup. | HackerNoon. Retrieved 28 January 2023, from https://hackernoon.com/pros-cons-you-must-know-before-using-ruby-on-rails-for-your-startup-234ecd631aaf

Getting Started with Rails. (n.d.). Retrieved 28 January 2023, from Ruby on Rails Guides website: https://guides.rubyonrails.org/getting_started.html

Gibbs, N. (2023). *Rails Ruby Bench* [Ruby]. Retrieved from https://github.com/noahgibbs/rails_ruby_bench (Original work published 2016)

goruby. (2022). *Goruby* [Go]. GoRuby. Retrieved from https://github.com/goruby/goruby (Original work published 2014)

Haynes, D. (2019, July 8). Puma vs Unicorn vs Passenger: Ruby App Servers Compared |

Scout APM Blog. Retrieved 10 January 2023, from https://scoutapm.com/blog/which-ruby-app-server-is-right-for-you

International Organization for Standardization. (2012). *Information technology—Programming languages—Ruby* (No. ISO/IEC 30170:2012). Retrieved from https://www.iso.org/obp/ui/#iso:std:iso-iec:30170:ed-1:v1:en

Internet Engineering Task Force. (1981). *Transmission Control Protocol* (Request for Comments No. RFC 793). doi: 10.17487/RFC0793

jruby. (2023). *Jruby/jruby* [Ruby]. JRuby Team. Retrieved from https://github.com/jruby/jruby (Original work published 2009)

Kazakov, A. (2023). *Andrey-kazakov/P2NC.01.116.thesis* [Ruby]. Retrieved from https://github.com/andrey-kazakov/P2NC.01.116.thesis (Original work published 2023)

Klochkov, D., & Mulawka, J. (2021). Improving Ruby on Rails-Based Web Application Performance. *Information*, *12*(8), 319. doi: 10.3390/info12080319

Kokubun, T. (2019, December 26). JIT development progress at Ruby 2.7. Retrieved 20 March 2023, from Medium website: https://k0kubun.medium.com/jit-development-progress-at-ruby-2-7-d6dd62a8c76a

Kokubun, T. (2021, May 21). Ruby 3 JIT can make Rails faster. Retrieved 8 January 2023, from Medium website: https://k0kubun.medium.com/ruby-3-jit-can-make-rails-faster-756310f235a

Kokubun, T. (2023). *Railsbench* [Ruby]. Retrieved from https://github.com/k0kubun/railsbench (Original work published 2019)

Lai, H. (2012, February 21). Ruby Enterprise Edition 1.8.7-2012.02 released; End of Life

imminent. Retrieved 19 March 2023, from Phusion Blog website: https://old.blog.phusion.nl/2012/02/21/ruby-enterprise-edition-1-8-7-2012-02-released-end-of-life-imminent/

Lin, J. (2022). *ErRuby—An implementation of the Ruby language on Erlang* [Erlang]. Retrieved from https://github.com/johnlinvc/erruby (Original work published 2015)

MagLev. (n.d.). Retrieved 21 March 2023, from http://maglev.github.io/

Maintenance Policy for Ruby on Rails. (n.d.). Retrieved 10 January 2023, from Ruby on Rails Guides website: https://guides.rubyonrails.org/maintenance_policy.html

Majkowski, M. (2022, February 2). How to stop running out of ephemeral ports and start to love long-lived connections. Retrieved 15 April 2023, from The Cloudflare Blog website: http://blog.cloudflare.com/how-to-stop-running-out-of-ephemeral-ports-and-start-to-love-long-lived-connections/

Matsumoto, Y., Makarov, V., Patterson, A., & Sasada, K. (2018, April 12). *MJIT: A Method Based Just-in-time Compiler for Ruby* (J. Scheffler, Interviewer). Retrieved from https://blog.heroku.com/ruby-mjit

Matsumoto, Y., & Sasada, K. (2007). *The Ruby VM Serial Interview* (J. E. Gray, Interviewer). Retrieved from http://graysoftinc.com/the-ruby-vm-interview/the-ruby-vm-serial-interview

Matsumoto, Y., Sasada, K., & Patterson, A. (2016, November 10). *Ruby 3x3: Matz, Koichi, and Tenderlove on the future of Ruby Performance* (J. Scheffler, Interviewer). Retrieved from https://blog.heroku.com/ruby-3-by-3

McWherter, D. T., Schroeder, B., Ailamaki, A., & Harchol-Balter, M. (2004). Priority

mechanisms for OLTP and transactional Web applications. *Proceedings. 20th International Conference on Data Engineering*, 535–546. Boston, MA, USA: IEEE Comput. Soc. doi: 10.1109/ICDE.2004.1320025

Pavese, T. (2016, December 21). Unicorn vs Puma: Rails server benchmarks. Retrieved 10 January 2023, from Deliveroo.engineering website: https://deliveroo.engineering/2016/12/21/unicorn-vs-puma-rails-server-benchmarks.html

puma. (2023). *Puma: A Ruby Web Server Built For Parallelism* [Ruby]. Puma. Retrieved from https://github.com/puma/puma (Original work published 2011)

RailsGuides. (n.d.). The Asset Pipeline. Retrieved 17 January 2023, from Ruby on Rails Guides website: https://guides.rubyonrails.org/asset_pipeline.html

Reporting Performance Problems. (n.d.). Retrieved 22 April 2023, from https://www.graalvm.org/latest/reference-manual/ruby/ReportingPerformanceProblems/

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. *Proceedings of the ACM Annual Conference on - ACM '72*, *2*, 717–740. Boston, Massachusetts, United States: ACM Press. doi: 10.1145/800194.805852

Ruby on Rails. (n.d.). Retrieved 28 January 2023, from Ruby on Rails website: https://rubyonrails.org/

Ruby Releases. (n.d.). Retrieved 19 March 2023, from https://www.ruby-lang.org/en/downloads/releases/

Ruby version policy changes starting with Ruby 2.1.0. (n.d.). Retrieved 13 February 2023, from

https://www.ruby-lang.org/en/news/2013/12/21/ruby-version-policy-changes-with-2-1-0/

Sasada, K. (2005). YARV: Yet another RubyVM: innovating the ruby interpreter. *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 158–159. San Diego CA USA: ACM. doi: 10.1145/1094855.1094912

Sasada, K. (2019). Gradual write-barrier insertion into a Ruby interpreter. *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, 115–121. Phoenix AZ USA: ACM. doi: 10.1145/3315573.3329986

The top programming languages. (2022). Retrieved 28 January 2023, from The State of the Octoverse website: https://octoverse.github.com/2022/top-programming-languages

truffleruby. (2023). *Oracle/truffleruby* [Ruby]. Oracle. Retrieved from https://github.com/oracle/truffleruby (Original work published 2016)

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., … Lang, M. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications*, *11*(2), 233–247. doi: 10.1007/s11761-017-0208-y

LICENCE

Mina, Andrei Kazakov,

1. Annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose “Comparison of performance of Ruby interpreters in high-load Rails applications”, mille juhendaja on Andre Sääsk, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

/allkirjastatud digitaalselt/
Andrei Kazakov
12.05.2023