Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 947–952

DOI: 10.15439/2015F99
ACSIS, Vol. 5

Ruby Benchmark Tool using Docker

Richard Ludvigh, Tomáš Rebok
Faculty of Informatics, Masaryk University
Botanická 68a, 60200, Brno, Czech Republic
Email: 409737@mail.muni.cz, xrebok@fi.muni.cz

Václav Tunka, Filip Nguyen
Red Hat Czech, JBoss Middleware
Purkyňova 111, 61245 Brno, Czech Republic
Email: {vtunka,fnguyen}@redhat.com

***Abstract*—The purpose of this paper is to introduce and describe a new Ruby benchmarking tool. We will describe the background of Ruby benchmarking and the advantages of the new tool. The paper documents the benchmarking process as well as methods used to obtain results and run tests. To illustrate the provided tool, results that were obtained by running a developed benchmarking tool on existing and available official ruby benchmarks are provided. These results document advantages in using various Ruby compilers or Ruby implementations.**

I. Introduction

RUBY IS A PURE OBJECT-ORIENTED interpreted language. The language itself has three major implementations: MRI written in C, JRuby written in Java, and Rubinius written in Ruby. These are often compared in different ways such as usage, performance, and memory requirements. The non-functional attributes of the implementations vary significantly.

Our goal was to develop a benchmarking tool for these implementations that would wrap all of the provided versions and run benchmarks against them. It is important to mention that this paper does not focus on developing benchmarks, but merely on providing a tool capable of taking any kind of existing benchmark, running it against all available Ruby versions, and testing its usability.

To ensure complete isolation of all tested Ruby versions, we used Docker [2] to pack each configuration inside a Docker container. Docker is a open-source tool that enables a Linux application and its dependencies to be packaged as a lightweight container. The benchmarking tool was then developed to handle these containers as well as to validate their content (correct versions, compilation flags, compilers used). It is also responsible for running selected benchmarks and storing their results. In section III, we describe the basic practices used while developing the benchmark tool as well as all of its responsibilities, methods used to collect data, and available Ruby versions.

In our research, we used official available benchmarks from the Ruby repository[1] and a parallelism benchmark from the Rubinius repository[2] to point out some basic characteristics of Ruby implementations and their power. The tool was run on a baremetal and virtual server to provide results from both environments. In section III-D, we describe both environments and their configuration in detail.

The results we present are also available online. The results are in three main areas:

- MRI Versions overview - we have compared multiple MRI Ruby versions to determine the progress mainly in memory usage as new MRI (2.2.0) has announced a new garbage collection algorithm.
- A comparison of MRI compilers determined the differences in using different C compilers to compile Ruby (2.2.0 used in benchmarks). Our results proved that the widely used, also default for a dominant group of linux distributions, version 4.8 of GCC is the best choice for prime performance.
- Running benchmarks on different Ruby implementations not only proved that MRI is best for short single-threaded executions (these tests were extremely short prohibiting to start the just-in-time compilers for JRuby and Rubinius) while JRuby and Rubinius are better for longer or multi-thread runs, but it also showed the progression and power of JRuby in handling parallel tasks and improving in performance from version to version.

II. State of The Art

Ruby [1] is an interpreted object-oriented programming language which was designed and released by Yukihiro Matsumoto, known as Matz, in 1995. Ruby is a pure object-oriented language in which even values of primitive types (true, false, nil) are represented as objects. It is also suitable for functional programming and capable of powerful metaprogramming.

There are three major Ruby implementations.

The oldest - original implementation - is known as MRI ("Matz's Ruby Interpreter") or CRuby (since it is written in C). CRuby does support native threads, but it uses Global Interpreter Lock [3] (known as GIL) which allows data to be modified only one thread at the time, thus prohibiting true concurrency.

JRuby, as the title suggests, is a 100% Java implementation of Ruby. This allows for Ruby applications to be run on a Java Virtual Machine (JVM), thus utilizing its just-in-time (JIT) compiler, garbage collection, and mainly its concurrent threads. It also allows us to use any library that is compatible with JVM.

Access to the CERIT-SC computing and storage facilities provided under the programme Center CERIT Scientific Cloud, part of the Operational Program Research and Development for Innovations, reg. no. CZ. 1.05/3.2.00/08.0144, is greatly appreciated.

[1] https://github.com/ruby/ruby

[2] https://github.com/rubinius/rubinius-benchmark

Rubinius is the Ruby version of Python's PyPy. Much of its source is written in Ruby making it easier to understand. It also includes a just-in-time compiler on a virtual machine written in C++.

In December 2013, Sam Saffron published a call for an official long-running Ruby benchmark [4]. At the beginning of development, in November 2014, Ruby still had no long term running benchmarks like Pypy speed center[3] for Python. At that time, there was just one Ruby benchmarking suite that was used by the community to test different configurations and experiment with their performance. The Ruby Benchmark Suite [5] by Antonio Cangiano was developed between the years 2008 and 2013. It uses a host OS and installed rubies to perform tests. During the development of our benchmark tool, Rubyfy.ME, Guo Xiang Tan[4] presented his own benchmark tool in cooperation with Sam Saffron. This became the official ruby long-term running benchmark[5]. His tool uses a single docker image containing all tested Ruby versions. It also provides tests for individual commits in the official Ruby repository acting more like performance CI (Continuous Integration) solution.

III. Benchmark Tool

In this section we describe the benchmark tool that we have developed. The main aim during development was to ensure the separation of all Ruby versions so they did not share a library or resource that could affect the results. To ensure complete isolation, docker images were created for each Ruby version or compiler. Since this tool is written in Ruby, there are only two system requirements that are required to run the developed benchmarking tool:

- Ruby of version 2.0 or above
- Docker of version 1.0.1 or above

A. Docker integration

Docker [2] is an open-source program which is capable of packing linux applications and their dependences as a container. Container-based virtualization isolates applications from each other. It runs in userspace on the host operating system utilizing resource isolation and allocation benefits while being much more efficient.

The tool automatically downloads all required docker images and validates a correct Ruby presence. This means that the correct Ruby version, the correct compiler and its version, and the correct compilation flags are present in each docker image. It is also responsible for running benchmarks in correct containers.

Table I lists Ruby versions and implementations that are present in the benchmarking tool.

[3]http://speed.pypy.org/
[4]https://github.com/tgxworld
[5]http://rubybench.org/

B. Running Benchmarks

All benchmarks must be located inside exactly one subfolder in the benchmarks folder. This hierarchy allows to track the origin or divide the benchmarks into groups. There is one special category of benchmarks located inside the benchmarks/custom folder (we will call them custom benchmarks from now) which is handled differently compared to the others. By default, official Ruby benchmarks are present in the benchmarks/ruby-official.

Every benchmark that is not from a custom category is, before its execution, wrapped inside an additional code that captures all of the needed information. Memory usage is stored before code execution. Then, using the official benchmark library for Ruby, time consumption is tracked. At the end, garbage collection is triggered manually and memory usage is tracked once more. This way, the needed time, used memory, and total executable memory are tracked for each benchmark.

Custom benchmarks do not undergo the described flow. Code is executed without modification and standard and error output is captured, thus allowing us to store any kind of information.

Rubinius and JRuby implementations of Ruby contain a just-in-time compiler which results in slower code execution during program startup. For these versions, it is important to warm up the virtual machine (JVM for JRuby) by running the code multiple times or for a short period of time before actually benchmarking. There is still no support because benchmarks need to be adapted to run in loops in order to ensure warm-up. For now, we do not use suitable benchmarks so all benchmarks are run without a warm-up, resulting in slower times for JRuby and Rubinius. However, we plan to add this feature after we ensure enough benchmarks that are able to run in loops.

C. Storing and Publishing Results

Each run stores a record in a csv file named after the Ruby version used for that run. Each record contains information about the executable benchmark, Ruby version, compiler, and current time. Standard output and standard error output are stored for custom benchmarks while time and memory usage are stored for others.

After providing information about the web interface of this benchmark tool, the user is able to push results from csv files to Ruby on Rails application which provides complex results and graphs.

The feature to publish stored results allows us to build an easy and user friendly presentation. Using the highcharts[6] library, simple graphs were created to simplify collected results. These graphs were split into three main categories: an MRI versions comparison, an MRI C compilers comparison, and a Ruby implementations comparison. Overall, graphs were created for each category as well as for each benchmark.

This tool was developed for community use so its main characteristic is easy usability and extensibility allowing users

[6]http://www.highcharts.com/

TABLE I
RUBY VERSIONS AND IMPLEMENTATIONS

Implementation	Versions	Details
JRuby	9.0.0.0.pre1	OpenJDK 64-Bit Server, Virtual machine version 1.7.0_75-b13
JRuby	1.7.12, 1.6.8	OpenJDK 64-Bit Server, Virtual machine version 1.7.0_65
Rubinius	2.2.10, 2.3.0, 2.4.0, 2.4.1	
MRI Ruby	2.2.0	Multiple images with different compilers: GCC 4.8, GCC 4.9, Clang 3.3, Clang 3.4, Clang 3.5, all on both -O2 and -O3 flags
MRI Ruby	1.6.8 - 2.1.5 (12 versions)	All compiled using GCC 4.8 -O2

to easily add new benchmarks, test their own code, or even add more Ruby versions. This tool also comes with its own Rails web application which allows users to see their results in no time. However, this approach makes it harder to specify or focus on some deeper characteristics of Ruby. For example, the user is currently unable to set or change runtime flags, but we are planning to provide this feature in the near future.

D. Environment

Benchmarking was performed in two independent environments.

- Baremetal Ubuntu 14.04, kernel version 3.13.0-36-generic x86_64 GNU/Linux, with Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz, 2x 8GB Samsung SODIMM DDR3 Synchronous 1333 MHz RAM located on an Intel Emerald Lake motherboard.
- A virtual private server provided by CERIT-SC, configured to 16GB RAM and 8 virtual CPU running Debian 3.16.7-ckt4-3 bpo70+1. This virtual machine is hosted on 2x Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz (12 cores) 96 GiB DDR3 1333 MHz. Center CERIT-SC (CERIT Scientific Cloud) offers computing resources and also participates in research and development activities.

Each Ruby implementation was run with its default settings. We did not use any runtime command flags as there is no support for this feature yet. The only exception was made during a parallelism test when JRuby and Rubinius were run with JIT disabled.

This tool was developed as an open-source project and its sources are publicly available on Github[7].

IV. RESULTS

Using benchmarks from the official Ruby repository enables us to present results in the following categories:

- MRI Ruby Compilers - a comparison of Ruby compilers (Clang, GCC) tested on MRI Ruby Versions 2.2.0.
- Ruby Implementations - the difference between various implementations and between handling single-thread vs. multi-thread tasks.
- Ruby 2.2.0 Garbage collection - the progress of new incremental Garbage collection announced in Ruby 2.2.0.

Each benchmark was successfully run ten times to provide stable and usable results from both environments. During the benchmarking and development process, the most recent versions of selected implementations were: 2.2.0 for MRI, 2.4.1 for Rubinius, and 9.0.0.0.pre1 for JRuby.

A. MRI Ruby Compilers

As MRI Ruby (also called CRuby) is written in C, the choice of a C compiler and its compilation flags can affect the performance of a Ruby interpreter.

In December 2014, Peter Wilmott published an article [6] about MRI Ruby compilers. All of his tests were run on AWS from an m3.medium EC2 instance and he used Ruby version 2.1. Using benchmarking suite developed by Antonio Cangiano, his results show a great performance increase from GCC 4.8 to GCC 4.9 and he also pointed out that optimization on level 2 works better than on level 3.

Running all of the official Ruby benchmarks on the Ruby version 2.2.0 on both a baremetal and a virtual server provided by CERIT-SC group resulted in our results being different than those obtained by Peter. Both sets provided us with the same result (Fig. 1 and Fig. 2), thus showing that GCC 4.8 on optimization level 3 (the default shipped for Ubuntu 14.04) is still ahead by a small amount (1-2% faster than GCC 4.9 -O3). The results also provided that level 3 optimization is currently faster for Ruby 2.2.0 (an almost 4% speed increase from GCC 4.8 -O2 to GCC 4.8 -O3).

B. Ruby Implementations

The main difference between MRI Ruby and Rubinius, JRuby, is that MRI Ruby uses GIL (global interpreter lock) which makes single threaded tasks run faster, but does not permit real concurrency. This is why MRI Ruby is significantly better in overall results (official ruby benchmarks that are used for each category are single thread simple tests and often too short to start a JIT compiler on JRuby and Rubinius, thus making their results even worse) as is shown on Fig. 3 and Fig. 4

Although we used the parallelism benchmark provided in the Rubinius repository[8] to determine the differences in handling parallel tasks. In this benchmark we manually edited the tool to pass command line arguments disabling JIT compilers for JRuby and Rubinius (-X-C for JRuby and -Xint for Rubinius) because this benchmark is built to run multiple times and provide the best results gathered. This would allow the JIT compilers to activate and radically reduce the execution times.

[7]https://github.com/Ryccoo/rubyfy-me-docker-suite

[8]https://github.com/rubinius/rubinius-benchmark/blob/master/parallelism.rb

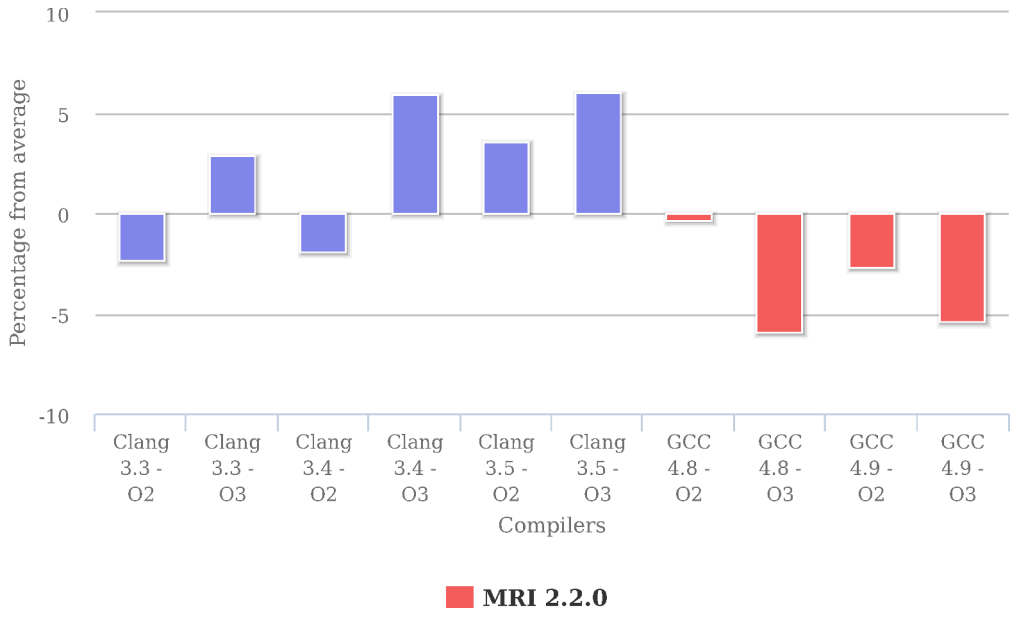

Fig. 1. Overall results for MRI compilers run on CERIT-SC virtual server

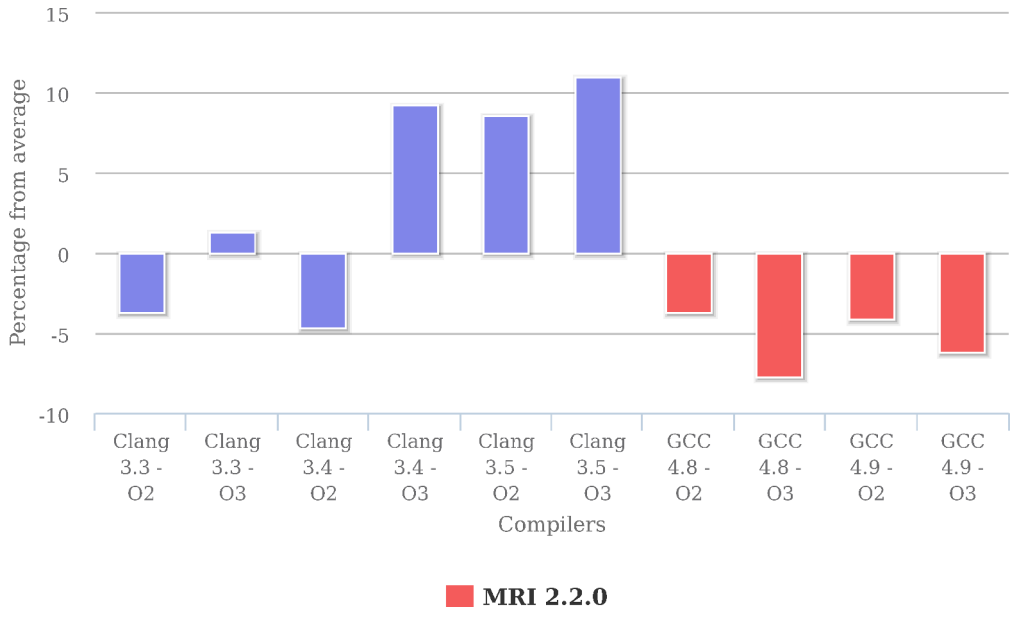

Fig. 2. Overall results for MRI compilers run on a baremetal Ubuntu machine

On Fig. 5 we can see the true power of JRuby parallelism as well as its progression.

The benchmark first computes the amount of work needed to keep the thread busy for two seconds, then runs the same amount of work on each of four threads. Because virtualization (the usage of virtual CPUs) makes it harder to compute and calibrate needed work amount, we present only results from the baremetal machine.

C. MRI 2.2.0 Incremental garbage collection

The MRI Ruby version 2.2.0 has announced new incremental garbage collection. From this version, symbols are also garbage collectable. Symbols are now divided into two categories: mortal and immortal. Immortal symbols are defined inside code while mortal symbols are created dynamically during execution. MRI Ruby 2.2.0 now collects mortal symbols allowing to free up more memory. We were able to watch the decrease of memory usage compared to the previous version as shown on Fig. 6.

These tests were also run separately and aimed at the last

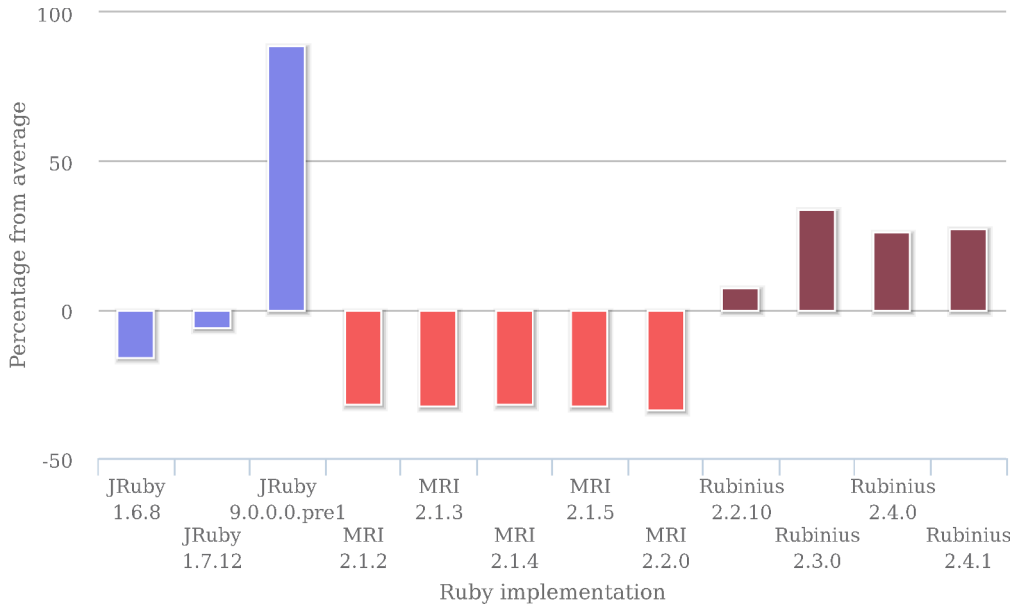

Fig. 3. Overall time performance on different Ruby implementations tested on a baremetal Ubuntu machine

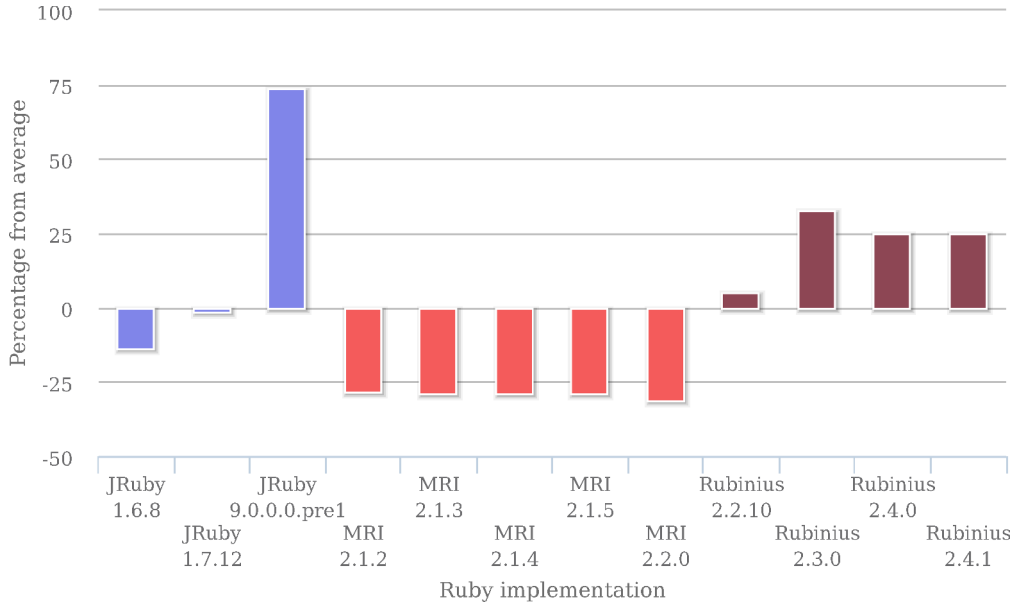

Fig. 4. Overall time performance on different Ruby implementations tested on a CERIT-SC virtual server

Ruby versions which confirmed the gap between ruby 2.1.x and 2.2.0 as shown on Fig. 7.

V. Conclusion

Our results show that in the case of compiling the newest version of MRI Ruby (currently 2.2.0 during benchmarking) for normal, non-experimental or special use cases, the choice of GCC 4.8 with an O3 flag will provide the best available performance. While this version of a GCC compiler is the default for the predominant group of linux distributions, there is no need to make any changes during Ruby installation.

MRI Ruby is the best choice for computing single threaded simple and short tasks while the GIL (Global Interpreter Lock) is prohibiting to starting real concurrency. This is when the choice of other implementations like JRuby and Rubinius is important. As shown on Fig. 5, JRuby provides us with the best concurrency. This ability increases from version to version. Therefore, JRuby is the best choice for big cloud computations offering multiple virtual CPUs.

It is important to remind that used benchmarks and benchmarking techniques did not allow the warm-up required by JIT compilers, thus results with proper and long enough warming

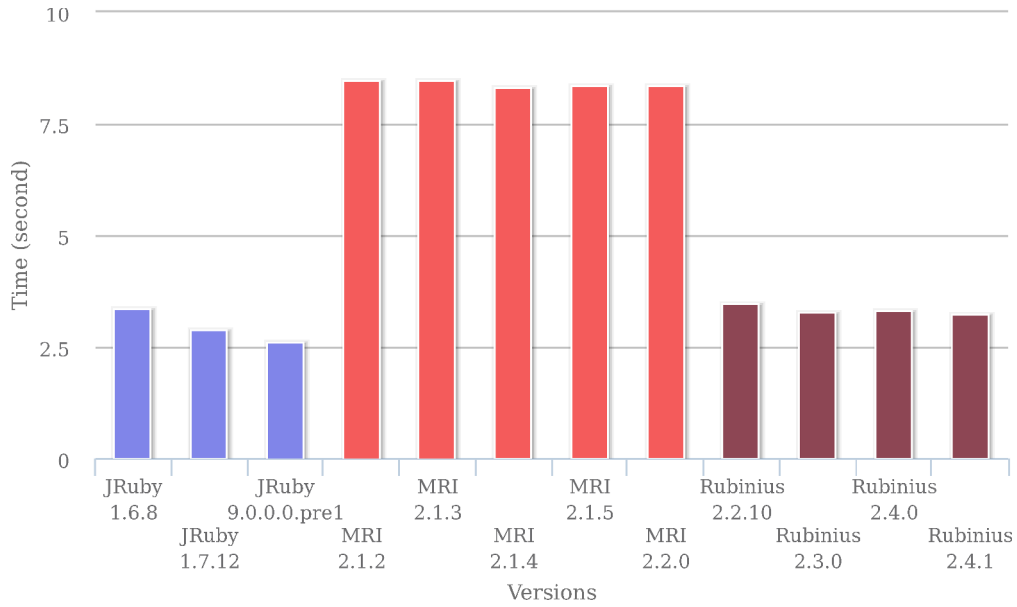

Fig. 5. Rubinius parallelism benchmark - running 4 parallel threads

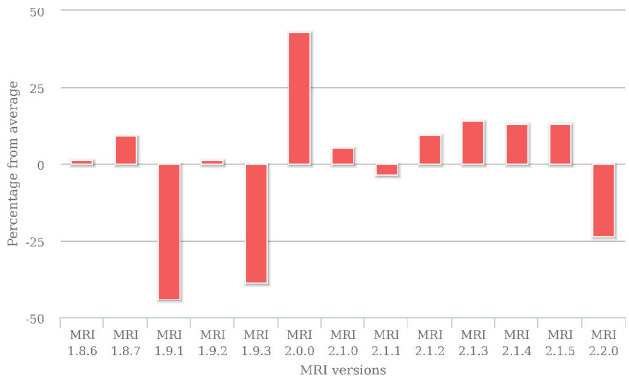

Fig. 6. Memory usage difference from the average

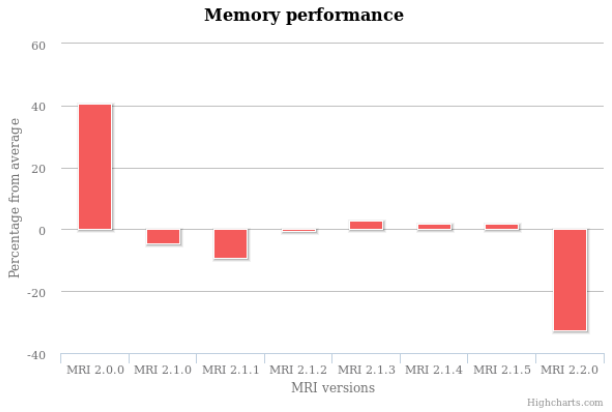

Fig. 7. Difference from average memory usage on recent Ruby versions

could differ on JRuby and Rubinius.

The introduction of new garbage collection in MRI 2.2.0 with the ability to collect mortal symbols can fix a common programmer mistake as a side effect. The problem occurred when the program was converting user inputs to symbols. These symbols were not garbage collectable and the program often drained the entire system memory and ended up freezing. Symbols converted from MRI 2.2.0 are tagged as mortal and garbage collected after being unused, thus not draining the entire memory.

REFERENCES

[1] Hirschfeld, Robert, and Kim Rose, eds. Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008, Proceedings. Vol. 5146. Springer Science & Business Media, 2008.

[2] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. Linux J. 2014, 239, pages.

[3] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. 2014. Eliminating global interpreter locks in ruby through hardware transactional memory. In Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '14). ACM, New York, NY, USA, 131-142.

[4] Call for official long running Ruby benchmark, http://samsaffron.com/archive/2013/12/11/call-to-action-long-running-ruby-benchmark

[5] Antonio Cangiano Ruby Benchmark, https://github.com/acangiano/ruby-benchmark-suite

[6] MRI Ruby Compilers Benchmark, https://www.p8952.info/ruby/2014/12/12/benchmarking-ruby-with-gcc-and-clang.html