Ruby Language Optimization Techniques

NICHOLAS BENDER, Boise State University
BEN NEELY, Boise State University
JOHN OTANDER, Boise State University

The Ruby programming language has experienced a recent period of intense adoption and growth due to its excellent speed of iteration, elegant syntax, and passionate community. Additionally, the popular web framework, Ruby on Rails, has given the Ruby language exceptional legitimacy, especially in the prototyping, startup space. It is a tool that emphasizes developer happiness, productivity, and places the responsibility of program in the developer's hands. This gives the language a lot of power, but can serve as a double-edged sword. When leveraged incorrectly, projects can swiftly become inefficient and unmaintainable. Additionally, allowing this flexibility has serious implications with memory management, efficiency, and execution times.

While support is growing steadily for the language, it is largely dismissed as not having effective scalability, and having far slower runtimes than more compiled, strongly-typed languages. In this article, we propose that many sophisticated techniques exist to enhance Ruby's performance both in using existing runtimes to compile ruby to statically typed languages, and in using common anti-patterns to improve performance natively. Through experimentation and thorough research we conclude that Ruby performs competitively against it's similar scripting language counterparts, and can see large increases in many cases.

Categories and Subject Descriptors: D.2.3 [Coding Tools and Techniques]: Object-oriented programming, B.6.3 [Design Aids]: Optimization
General Terms: Optimization, Algorithms, Performance
Additional Key Words and Phrases: Ruby, Web Development, JRE, C++, C

INTRODUCTION

Ruby is an object oriented, dynamically-typed, high-level scripting language. It is a programming language that was written for humans and just happens to run on computers. It's intended to promote developer happiness through simplicity, elegant libraries, and terse, readable syntax. Ruby also uses duck typing, meaning type is determined through methods and properties. With each of these techniques and language features there exist certain sacrifices. In this exploration we will conclude that the best practices for stable, performant Ruby programs exist by utilizing the newest versions of the core language properly.

In recent years, the Ruby programming language has grown its community and established itself as a valuable, popular tool for many tasks [O'Donoghue, 2014]. The success of Ruby on Rails as a prototyping framework, as well as a full-stack solution for some larger companies, has brought forth a myriad of techniques to ensure that the language's speed differences compared to similar languages are minimal. Ruby's slower performance, as compared to C or Java, is attributed to interpreted execution, dynamic typing, meta-programming support, and the Global Interpreter Lock [Odaira, Castanos, and Tomari, 2014]. This increase in popularity has caused a large number independent optimization efforts to arise from large corporations such as IBM and AT&T, as well as efforts from the Ruby open-source community.

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
> *- Heim, Michael (2007).*

1. MRI (> 1.9)

The MRI is short for Matz's Ruby Interpreter, which is sometimes also referred to as CRuby. The MRI is named after Yukihiro Matsumoto, the chief designer of the Ruby language. The original MRI was the runtime environment from Ruby's inception to 1.8.7.

1.1 Program Execution at a High Level

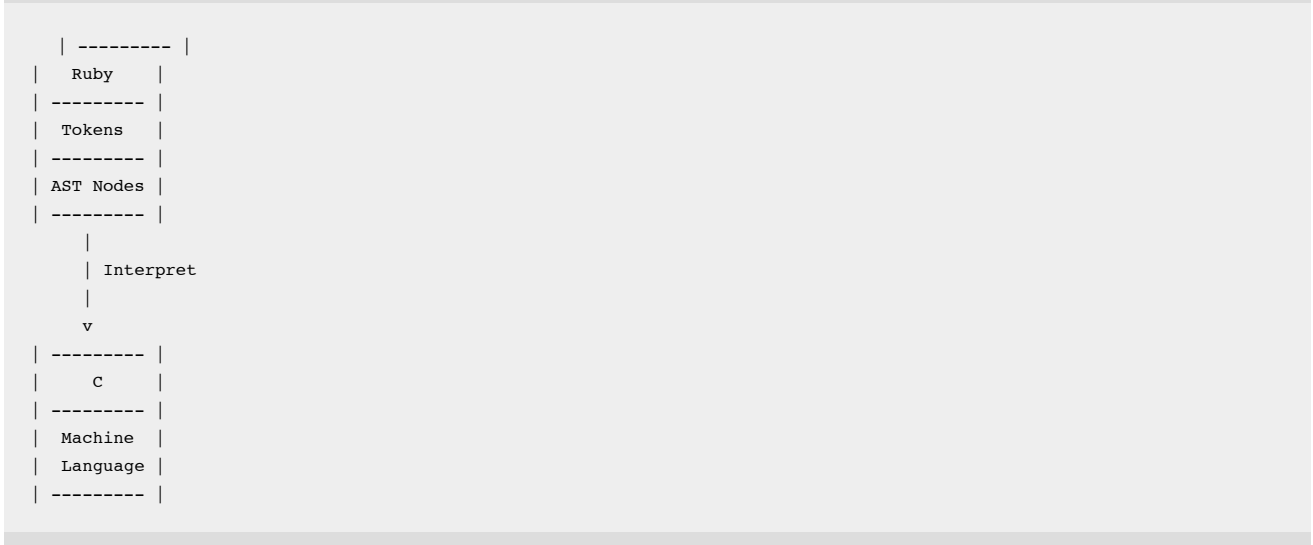

Fig. 1. Ruby Program Execution

A Ruby script undergoes a tokenization step, which is then parsed into an Abstract Syntax Tree. The Ruby C code (MRI), reads and executes the AST. Note that there is no compilation or

translation step.

1.2 Performance

Since there isn't a bytecode compilation step, the execution of Ruby programs requires walking the MRI's internal Abstract Syntax Tree. This slows the execution speed significantly because it's more costly to interpret the AST data structure during runtime.

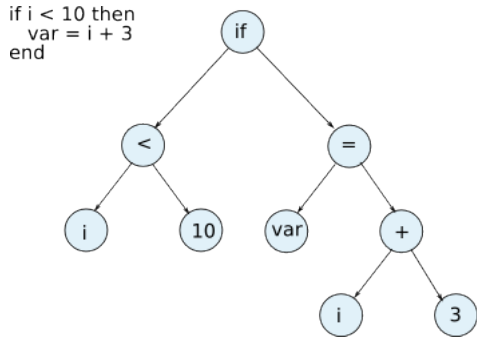

Fig. 2. Abstract Syntax Tree http://edwinmeyer.com/Release_Integrated_RHG_09_10_2008/intro.html

1.3 Optimizations

Use receiver methods whenever possible because it avoids the allocation of a copied string.

```
   2.1.1 :003 > str = "A string.\n"
 => "A string.\n"
2.1.1 :004 > str2 = str
 => "A string.\n"
2.1.1 :005 > str.chomp!
 => "A string."
2.1.1 :006 > str2
 => "A string."
2.1.1 :007 >
```

Fig. 3. Receiver modifying methods vs receiver duplicating methods

1.4 Summary

The initial implementation of the MRI is one of the primary reasons that Ruby get its "bad wrap" for code execution speed.

2. JRUBY

2.1 Purpose

Jruby endeavors to solve many Ruby performance issues by eliminating the standard interpreter

and instead taking ruby syntax and compiling as much of the core libraries as possible to Java bytecode. Current versions of JRuby support both just-in-time compilation as well as ahead-of-time compilation to Java bytecode. In using these various stages of bytecode in addition to some portions of the standard interpreter, this allows for several advantages over the standard interpreter.

One of the more obvious improvements is the ability to call and use standard Java libraries and classes from within ruby projects. For larger organizations already using Java for core library support, this allows for improved flexibility of the development environment.

2.2. Performance

In 2007, JRuby's overall performace was compared with Ruby 1.8.5, the Yarv interpreter (now merged into Ruby's official interpreter), and Rubinius. In it, only 10% of tests performed had JRuby outperforming standard Ruby. These speed enhancements, however, still managed to run all Ruby benchmarks without timing out or producing an error, a claim that no other non-standard Ruby implementation could make.

Recent benchmarks performed in 2014 between the latest implementations of JRuby and Ruby are comparable to standard Ruby. While some benchmarks provided an optimized runtime, the increased memory overhead of JRuby (>10x) makes scaling ruby applications problematic.

In addition to JRuby's memory woes, the biggest performance downside of JRuby comes from the speed of initializing the JVM to begin with. A simple ruby script that would take the MRI a fraction of a second to run would require several additional seconds just due to JVM launch times.

2.3 Lack of C Support

While JRuby allows for enhanced support and compatibility with Java libraries and applets, the majority of Ruby users (especially those using Ruby on Rails) are used to using libraries that contain native C support. In choosing to support Java, JRuby forces the incompatibility with native C extensions. Most notably are a variety of database interfaces and web servers.

2.4. Development Lag

Due to JRuby's implementation being dependent on Ruby releases prior to implementation and support, this has created an unfortunately long lag time, with the most recent release of JRuby only supporting Ruby version 1.9.3, which was initially released in 2011.

2.5 Summary

While JRuby does offer some improved benchmark performance in a minority of cases, the slow development cycle and potential for a massive increase to memory footprint make it an unsuitable option for pure ruby development stacks.

3. Rubinius

3.1 Purpose

Rubinius is an implementation of the Ruby programming language and includes a bytecode virtual machine, Ruby syntax parser, bytecode compiler, generational garbage collector, just-in-time (JIT) native machine code compliler, and Ruby Core and Standard Libraries. Rubinius is written using Ruby and C++.

3.2 History

Rubinius was originally created to be a Ruby virtual machine and runtime written in pure ruby. The current ruby interpreter is primarily writen in non-Ruby langauges such as C. From 2007 to 2013, the software company Engine Yard was a primary backer of Rubinius. During that time the focus of Rubinius evolved from creating a completely bootstrapped Ruby VM to instead offering an implementation of Ruby with increased performance. Under this new direction, Rubinius partially abandoned the idea of bootstrapping the Ruby VM in all Ruby code, and instead sought to use C++ to increase performance and establish Rubinius as the fastest Ruby implementation. Recently Rubinius has focused on supporting concurrency and multi-threading.

3.3 Performance

Rubinius initially achieved performance equal or slightly better to that of the Yarv interpreter. However, in recent years the MRI interpreter has consistently out performed Rubinius on most benchmark tests.

Rubinius consistently benchmarks as one of the slowest modern implementations of the Ruby language.

3.4 Concurrency

Rubinius does outperform the MRI in threading and concurrency benchmark tests. As shown in the figure bellow, Rubinius (represented by rbx–2.0.0) has a nontrivial advantage over MRI and

other Ruby implementations when exciting multithreaded code.
Rubinius is unique amongst Ruby implementations in that it does not have Global Interpreter Lock (GIL). The GIL in all other Reuby implementation allows only one thread to execute at at a time, no matter how many processor cores are available. Not implementing the GIL gives Rubinius the ability to support true threading

3.5 Summary

Rubinius' development has been spotty, depending heaving on a few developers and a few corporate sponsors. As a result Rubinius has constantly shifted focus. Rubinius currently offers a significant advantage over other Ruby interpreters only with regards to programming involving threading and concurrency. For all other uses, the standard MRI Ruby interpreter is faster and more consistently supported.

4. YARV

4.1 Background

```
CODE => TOKENIZATION => PARSE TREE => COMPILATION => YARV INSTRUCTIONS
```

When a Ruby program is executed, it first tokenizes the program. This means that the contents are converted into a collection of tokens with associated types. Ruby uses the LALR (Look-Ahead Left Reversed Rightmost Derivation) Parser to apply meaning to the tokens and construct the Abstract Syntax Tree. The compilation step was introduced with Ruby 1.9, and is where the YARV (Yet Another Ruby Virtual Machine) comes into play. It translates the code into bytecode, or YARV instructions.

```
  ~|||$ irb
2.1.1 :001 > code = <<CODE
2.1.1 :002"> puts 1 + 2
2.1.1 :003"> CODE
 => "puts 1 + 2\n"
2.1.1 :004 > puts RubyVM::InstructionSequence.compile(code).disasm
== disasm: <RubyVM::InstructionSequence:<compiled>@<compiled>>==========
0000 trace            1                                               (   1)
0002 putself
0003 putobject_OP_INT2FIX_O_1_C_
0004 putobject        2
0006 opt_plus         <callinfo!mid:+, argc:1, ARGS_SKIP>
0008 opt_send_simple  <callinfo!mid:puts, argc:1, FCALL|ARGS_SKIP>
0010 leave
 => nil
```

Fig. 4. YARV instructions for a simple program

The introduction of the compilation step and YARV have significantly helped the execution speed of Ruby programs. However, there's always room for more improvements.

4.2 Purpose

The Ruby MRI is short for Matz's Ruby Interpreter, and is the reference implementation for the Ruby programming language. It was released to the public in 1995, and is still actively developed, with the latest stable build being Ruby 2.1.1.

The YARV is an interpreter developed by Koichi Sasada that's also known as the KRI. It was developed in order to reduce the execution time of Ruby programs, and was very successful. As a result, YARV was merged into Ruby 1.9.0 and has replaced the MRI.

As the default interpreter for the Ruby programming language, the MRI has received it's fair share of criticism, primarily due to it's execution speeds and memory consumption. However, recent Ruby versions have seen significant enhancements, and is on par with similar scripting languages like Python. Not to mention the fact that some comparisons have the audacity to compare a compiled language to a scripting language, which is apples to oranges. Developers typically choose Ruby for it's ease of writing/prototyping, understanding the fact that its execution time will always be significantly slower than its compiled counterparts.

That being said, there are numerous methods and best practices that developers can follow in order to ensure that they're avoiding unnecessary bottlenecks.

4.3 Performance Out of the Box

Thanks to the introduction of YARV, vanilla Ruby, on a single thread, has the ability to outperform other alternative Ruby implementations. Consider the following figure, that measures Rails requests per second.

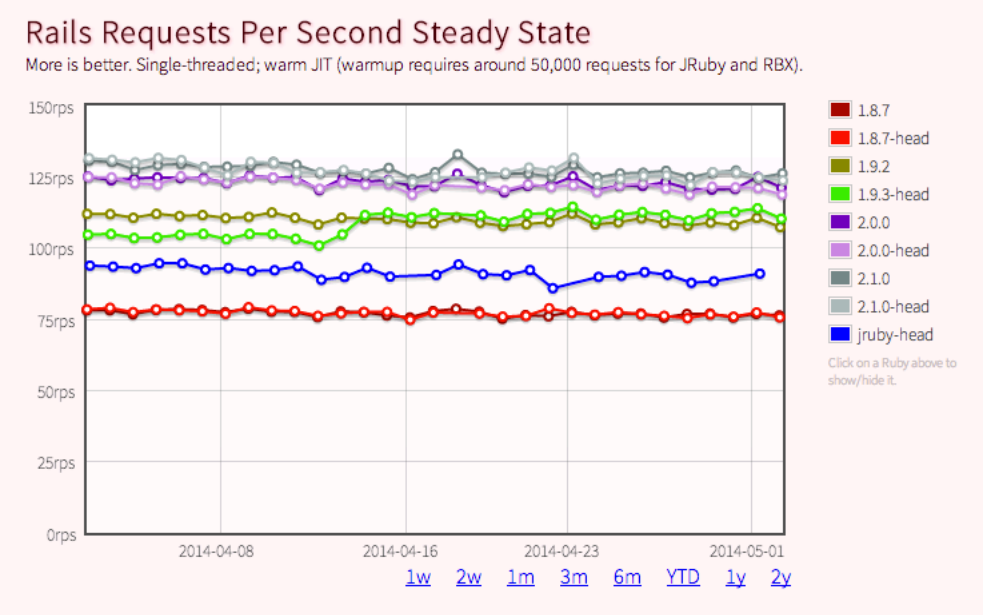

Fig. 5. Rails requests per second.

4.4 Global Interpreter Lock

When attempting to optimize execution speed, threads are often utilized in order to process tasks concurrently. This is a feature that Ruby supports, too. However, the MRI/YARV incorporates a Global Interpreter Lock, or GIL, that doesn't permit any true concurrency. A GIL refers to an interpreter thread that doesn't allow code that isn't thread safe to share itself with other threads. This results in little, to no, actual gain in speed when running threads on a multiprocessor machine.

The primary reason is that the GIL is used to avoid race conditions within C extensions. There are also thread safety reasons, too. Parts of Ruby aren't thread safe (Hash), and numerous C libraries that are wrapped by Ruby's internals. Additionally, the GIL is integral to data integrity, because it ensures that the developer doesn't write any unsafe threading code.

This, interestingly enough, runs contrary to the fundamental principles of the Ruby language, where all the responsibility is laid on the developer. Ruby allows the developer to have the ultimate freedom without hand holding, yet the GIL is just that, hand holding.

The GIL isn't going anywhere. It is deeply intertwined with Ruby and its internals, and many influential Ruby-core figures don't plan on removing the GIL anytime in the near future. Though, this doesn't mean the concurrency can't be achieved.

Though, you can sidestep the GIL with multiple virtual machines. Sasada Koichi has proposed a Multiple VM (MVM) solution, which is currently being developed. This would consist of multiple virtual machines, running their own processes, and communicate via sockets.

Granted, this is a drastic step away from typical threading, but some proponents believe that traditional threading isn't necessarily the correct paradigm to follow. Especially considering the fact the Ruby leverages green threads above the GIL rather than talking to the OS directly.

> Nevertheless, you're right the GIL is not as bad as you would initially think: you just have to undo the brainwashing you got from Windows and Java proponents who seem to consider threads as the only way to approach concurrent activities. Just because Java was once aimed at a set-top box OS that didn't support multiple address spaces, and just because process creation in Windows used to be slow as a dog, doesn't mean that multiple processes (with judicious use of IPC) aren't a much better approach to writing apps for multi-CPU boxes than threads.

4.5 Simple Code Enhancements

String interpolation is significantly more performant than concatentation because it doesn't need to allocate new strings, it just modifies a single string in place.

```
   require 'benchmark'

concat_time = Benchmark.measure do
  20000000.times do
    str = 'str1' << 'str2' << 'str3'
  end
end

# => #<Benchmark::Tms:0x007fdf9ba49ea8 @label="", @real=79.152523, @cstime=0.0, @cutime=0.0, @stime=0.04000000000000001, @utime=79.1

interp_time = Benchmark.measure do
  20000000.times do
    str = "#{str1}#{str2}#{str3}"
  end
end

# => #<Benchmark::Tms:0x007fdf9b990bd8 @label="", @real=22.713976, @cstime=0.0, @cutime=0.0, @stime=0.009999999999999995, @utime=22.
```

Fig. 6. Interpolation vs concatenation of Ruby Strings

The collect|map methods with blocks are faster because it returns a new array rather than an enumerator. This can be leveraged to increase speed when compared to Symbol.to_proc implementations. Though, the latter is typically much more preferable to read. The reason that the Symbol.to_proc is slower is because to_proc is called on the symbol to perform the following conversion:

```
   :method.to_proc
# => -> x { x.method }
```

```
fake_data = 20.times.map { |t| Fake.new(t) }

proc_time = Benchmark.measure do
  200000000.times do
    fake_data.map(&:id)
  end
end

#  => #<Benchmark::Tms:0x007fdf9b8b0498 @label="", @real=491.332415, @cstime=0.0, @cutime=0.0, @stime=4.8, @utime=426.06999999999994
```

```
  block_time = Benchmark.measure do
  200000000.times do
    fake_data.map { |d| d.id }
  end
end

# => #<Benchmark::Tms:0x007fdf9b931d40 @label="", @real=431.731424, @cstime=0.0, @cutime=0.0, @stime=2.66, @utime=416.21000000000004
```

```
  collect_time = Benchmark.measure do
  200000000.times do
    fake_data.collect { |d| d.id }
  end
end

# => #<Benchmark::Tms:0x007fdf9b821518 @label="", @real=386.234513, @cstime=0.0, @cutime=0.0, @stime=1.1800000000000006, @utime=384.
 :037 >
```

Fig. 7. Procs vs Blocks vs Collects

There are also garbage collection modifications that can be made in order to further optimize Ruby execution speed for most systems.

```
  # This is 60(!) times larger than default
RUBY_HEAP_MIN_SLOTS=600000

# This is 7 times larger than default
RUBY_GC_MALLOC_LIMIT=59000000

# This is 24 times larger than default
RUBY_HEAP_FREE_MIN=100000
```

Fig. 8. Garbage Collection Modification

4.5 Use Unicorn

For Ruby on Rails web applications, a server typically runs on a single process, which means that every request is processed one at a time. This can create a significant bottle neck in your application. Fortunately, there are libraries to incorporate concurrency in your application. One of which is Unicorn.

Unicorn uses Unix forks within a dyno (web worker) to create multiple instances of itself. Now, there are multiple OS instances that can all respond to requests, and complete tasks

concurrently. This results in smaller queues, quicker responses, and a faster web application as a whole. The only drawback is memory usage, which can grow to large sizes. Though, with decreasing hardware costs, this becomes a worthwhile expenditure to ensure quick development time for the software components. This also doesn't require thread safe code, since each worker is a self-sufficient clone of the parent.

Ruby 2.0 makes process forking even more efficient with Unicorn because it implements Copy-on-Write (CoW), which means that a parent and child share physical memory until a write needs to be made. This is a very efficient sharing of resources that can drastically reduce memory use.

Sometimes, there are still issues with memory leakage, which occurs when workers get stuck or timeout. With the inclusion of a gem, and a small snippet of code that's included below, these edge cases are covered.

```
   if ENV['RAILS_ENV'] == 'production'
  require 'unicorn/worker_killer'

  max_request_min =  500
  max_request_max =  600

  # Max requests per worker
  use Unicorn::WorkerKiller::MaxRequests, max_request_min, max_request_max

  oom_min = (240) * (1024**2)
  oom_max = (260) * (1024**2)

  # Max memory size (RSS) per worker
  use Unicorn::WorkerKiller::Oom, oom_min, oom_max
end

require ::File.expand_path('../config/environment',  __FILE__)
run YourApp::Application
```

Fig. 9. Example Unicorn Implementation

CONCLUSIONS

In this article, we examined a number independent Ruby optimization efforts. Each of these efforts seek to achieve performance improvements through a variety of techniques. In our examination we've determined that for each of these techniques there are certain sacrifices, that outweigh the marginal benefits are gained. Unless a particular feature is needed (such as full threading support or inline Java) the best practices for stable, performant Ruby code exist by utilizing the newest versions of the core language.

ACKNOWLEDGMENTS

The authors would like to thank Douglas Wiegley. He knows what he did.

REFERENCES

ROBERT O'DONOGHUE. 2014. Careers Close-up: programmers and software engineers. (March 2014). Retrieved March 31, 2014 http://www.siliconrepublic.com/careers/item/36001-crs-cls-up

REI ODAIRA, JOSE G. CASTANOS, HISANOBU TOMARI. 2014. Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory. PPoPP'14, February 15–19 2014, Orlando, FL, USA. DOI: http://dx.doi.org/10.1145/2555243.2555247

ANTONIO CANGIANO. 2007. The Great Ruby Shootout (December 2007). Retrieved March 31, 2014 http://programmingzen.com/2007/12/03/the-great-ruby-shootout/

PAT SHAUGHNESSY. 2014. Ruby Under a Microscope: An Illustrated Guide to Ruby Internals

BUSSINK DIRKJAN. Rubinius - Tales from the Trenches of Developing a Ruby implementation, Barcelona Ruby Conference, 2012.

NUTTER CHARLES. Why JRuby?, Aloha Ruby Conf, 2012.

SASADA KOICHI. YARV: Yet Another RubyVM-The Implementation and Evaluation. Transactions of Information Processing Society of Japan. Volume 47. 2006. Pages 57–73.

SASADA KOICHI. YARV: yet another RubyVM: innovating the ruby interpreter. OOPSLA '05 Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Pages 158–159.

SHAUGHNESSY PAT. Visualizing Garbage Collection in Rubinius, JRuby and Ruby 2.0, Ruby Conference, 2013.

YUKIHIRO MATSUMOTO. 2010. From Lisp to Ruby to Rubinius.