Extracted from:

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

Alexander Dymo

edited by Michael Swaine

Ruby Performance Optimization

Why Ruby Is Slow, and How to Fix It

Alexander Dymo

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-069-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2015

It’s time to optimize.

This is what I think when my Heroku dyno restarts after logging an “R14 - Memory quota exceeded” message. Or when New Relic sends me another bunch of Apdex score alerts. Or when simple pages take forever to load and the customer complains that my application is too slow. I’m sure you’ve had your own “time to optimize” moments. And every time these moments occur, we both ask exactly the same question: “What can I do to make the code faster?”

In my career as a Ruby programmer I have learned that the immediate answer to this question is often “I don’t know.” I’ll bet that’s your experience, too. After all, you thought you were writing efficient code. What we typically do then is to skip optimization altogether and resort to caching and scaling. Why? Because we don’t immediately see how to improve the code. Because conventional wisdom says optimization is hard. And because caching and scaling are familiar to seasoned Ruby developers. In most cases you only need to configure some external tool and make minimal changes to the code, and voilà! Your code is faster.

But there is a limit to what caching and scaling can do for you. One day my company discovered that Hirefire, the automated scaling solution for Heroku, scaled up the number of Heroku web dynos to 36 just to serve a meager five requests per minute. We would have to pay $3,108 per month for that. And our usual bill before was $228 for two web dynos and one worker. Whoa, why did we have to pay almost fifteen times more? It turned out there were two reasons for that. First, web traffic increased. Second, our recent changes in the code made the application three times slower. And our traffic kept increasing, which meant that we’d have to pay even more. Obviously, we needed a different approach. This was a case where we hit a limit to scaling and had to optimize.

It is also easy to hit a limit with caching. You can tell that you need to stop caching when your cache key gets more and more granular.

Let me show you what I mean with a code snippet from a Rails application of mine:

```
cache_key = [@org.id, @user.id,
  current_project, current_sprint, current_date,
  @user_filter, @status_filter,
  @priority_filter, @severity_filter, @level_filter]

cache(cache_key.join("_")) do
  render partial: 'list'
```

```
end
```

Here my cache key consists of ten parts. You can probably guess that the likelihood of hitting such a granular cache is very low. This is exactly what happened in reality. At some point my application started to spend more resources (either memory for Memcached or disk space) for caching than for rendering. Here's a case where further caching would not increase performance and I again had to optimize.

So have I convinced you of the need to optimize? Then let's learn how.

Here's when most sources on performance optimization start talking about execution time, profilers, and measurements. The hard stuff. We'll do our own share of profiling and measuring, but let's first step back and think about what exactly we need to optimize. Once we understand what makes Ruby slow, optimization stops being a search for a needle in a haystack with the profiler. Instead it can become almost a pleasing task where you attack a specific problem and get a significant performance improvement as the reward.

What Makes Ruby Code Slow

To learn what makes Ruby code fast, we must understand what makes Ruby code slow.

If you've done any performance optimization in the past, you probably think you know what makes code slow. You may think that even if you haven't done performance optimization. Let me see if I can guess what you think.

Your first guess is algorithmic complexity of the code: extra nested loops, computations, that sort of stuff. And what would you do to fix the algorithmic complexity? Well, you would profile the code, locate the slow section, identify the reason for the slowness, and rewrite the code to avoid the bottleneck. Rinse and repeat until fast.

Sounds like a good plan, right? However, it doesn't always work for Ruby code. Algorithmic complexity can be a major cause for performance problems. But Ruby has another cause that developers often overlook.

Let me show you what I'm talking about. Let's consider a simple example that takes a two-dimensional array of strings and formats it as a CSV.

Let's jump right in. Key in or download this simple program.

chp1/example_unoptimized.rb

```
require "benchmark"

num_rows = 100000
```

```
num_cols = 10
data = Array.new(num_rows) { Array.new(num_cols) { "x"*1000 } }

time = Benchmark.realtime do
  csv = data.map { |row| row.join(",") }.join("\n")
end

puts time.round(2)
```

We'll run the program and see how it performs. But before that we need to set up the execution environment. There are five major Ruby versions in use today: 1.8.7, 1.9.3, 2.0, 2.1, and 2.2. These versions have very different performance characteristics. Ruby 1.8 is the oldest and the slowest of them, with a different interpreter architecture and implementation. Ruby 1.9.3 and 2.0 are the current mainstream releases with similar performance. Ruby 2.1 and 2.2 are the only versions that were developed with performance in mind, at least if we believe their release notes, and thus should be the fastest.

It's hard to target old software platforms, so I'll make a necessary simplification in this book. I will neither write examples nor measure performance for Ruby 1.8. I do this because Ruby 1.8 is not only internally different, it's also source-incompatible, making my task extremely complicated. However, even if you have a legacy system running Ruby 1.8 with no chance to upgrade, you can still use the performance optimization advice from this book. Everything I describe in the book applies to 1.8. In fact, you might even get more improvement. The old interpreter is so inefficient that any little change can make a big difference. In addition to that I will give 1.8-specific advice where appropriate.

The easiest way to run several Rubys without messing up your system is to use rbenv or rvm. I'll use the former in this book. Get rbenv from https://github.com/sstephenson/rbenv. Follow the installation instructions from README.md. Once you install it, download the latest releases of Ruby versions that you're interested in. This is what I did; you may want to get more recent versions:

```
$ rbenv install -l
  ...
  1.9.3-p551
  2.0.0-p598
  2.1.5
  2.2.0
  ...
$ rbenv install -k 1.9.3-p551
$ rbenv install -k 2.0.0-p598
$ rbenv install -k 2.1.5
$ rbenv install -k 2.2.0
```

Note how I install Ruby interpreters with the k option. This keeps sources in rbenv's directory after compilation. In due time we'll talk about the internal Ruby architecture and implementation, and you might want to have a peek at the source code. For now, just save it for the future.

To run your code under a specific Ruby version, use this:

```
$ rbenv versions
* system (set by /home/user/.rbenv/version)
  1.9.3-p551
  2.0.0-p598
  2.1.5
  2.2.0
$ rbenv shell 1.9.3-p551
$ ruby chp1/example_unoptimized.rb
```

To get a rough idea of how things perform, you can run examples just one time. But you shouldn't make comparisons or draw any conclusions based on only one measurement. To do that, you need to obtain statistically correct measurements. This involves running examples multiple times, statistically post-processing the measurement results, eliminating external factors like power management on most modern computers, and more. In short, it's hard to obtain truly meaningful measurement. We will talk about measurements later in Chapter 7, *Measure*, on page ?. But for our present purposes, it is fine if you run an example several times until you see the repeating pattern in the numbers. I'll do my measurements the right way, skipping any details of the statistical analysis for now.

OK, so let's get back to our example and actually run it:

```
$ rbenv shell 1.9.3-p551
$ ruby example_unoptimized.rb
9.18
$ rbenv shell 2.0.0-p598
$ ruby example_unoptimized.rb
11.42
$ rbenv shell 2.1.5
$ ruby example_unoptimized.rb
2.65
$ rbenv shell 2.2.0
$ ruby example_unoptimized.rb
2.43
```

Let's organize the measurements in a tabular format for easy comparison. Further in the book, I'll skip the session printouts and will just include the comparison tables.

	1.9.3	2.0	2.1	2.2
Execution time	9.18	11.42	2.65	2.43

What? Concatenating 100,000 rows, 10 columns each, takes up to 10 seconds? That's way too much. Ruby 2.1 and 2.2 are better, but still take too long. Why is our simple program so slow?

Let's look at our code one more time. It seems like an idiomatic Ruby one-liner that is internally just a loop with a nested loop. The algorithmic efficiency of this code is going to be O(n m) no matter what. So the question is, what can we optimize?

I'll give you a hint. Run this program with garbage collection disabled. For that just add a `GC.disable` statement before the benchmark block like this:

chp1/example_no_gc.rb
```
require "benchmark"

num_rows = 100000
num_cols = 10
data = Array.new(num_rows) { Array.new(num_cols) { "x"*1000 } }

GC.disable
time = Benchmark.realtime do
  csv = data.map { |row| row.join(",") }.join("\n")
end

puts time.round(2)
```

Now let's run this and compare our measurements with the original program.

	1.9.3	2.0	2.1	2.2
GC enabled	9.18	11.42	2.65	2.43
GC disabled	1.14	1.15	1.19	1.16
% of time spent in GC	88%	90%	55%	52%

Do you see why the code is so slow? Our program spends the majority of its execution time in the garbage collector—a whopping 90% of the time in older Rubys and a significant 50% of the time in modern versions.

I started my career as C++ developer. That's why I was stunned when I first realized how much time Ruby GC takes. This surprises even seasoned developers who have worked with garbage-collected languages like Java and C#. Ruby GC takes as much time as our code itself or more. Yes, Ruby 2.1 and later perform much better. But even they require half the execution time for garbage collection in our example.

What's the deal with the Ruby GC? Did our code use too much memory? Is the Ruby GC too slow? The answer is a resounding yes to both questions.

High memory consumption is intrinsic to Ruby. It's a side effect of the language design. "Everything is an object" means that programs need extra memory to represent data as Ruby objects. Also, slow garbage collection is a well-known historical problem with Ruby. Its mark-and-sweep, stop-the-world GC is not only the slowest known garbage collection algorithm. It also has to stop the application for the time GC is running. That's why our application takes almost a dozen seconds to complete.

You have surely noticed significant performance improvement with Ruby 2.1 and 2.2. These versions feature much improved GC, called restricted generational GC. We'll talk about what that means later in Chapter 10, *Tune Up the Garbage Collector*, on page ?. For now it's important to remember that the latest two Ruby releases are much faster thanks to the better GC.

High GC times are surprising to the uninitiated. Less surprising, but still important, is the fact that without GC all Ruby versions perform the same, finishing in about 1.15 seconds. Internally the Ruby VMs are not that different across the versions starting from 1.9. The biggest improvement relevant to performance is the restricted generational GC that came with Ruby 2.1. But that, of course, has no effect on code performance when GC is disabled.

If you're a Ruby 1.8 user, you shouldn't expect to get the performance of 1.9 and later, even with GC turned off. Modern Rubys have a virtual machine to execute precompiled code. Ruby 1.8 executes code in a much slower fashion by traversing the syntax tree.

OK, let's get back to our example and think about why GC took so much time. What did it do? Well, we know that the more memory we use, the longer GC takes to complete. So we must have allocated a lot of memory, right? Let's see how much by printing memory size before and after our benchmark. The way to do this is to print the process's RSS, or Resident Set Size, which is the portion of a process's memory that's held in RAM.

On Linux and Mac OS X you can get RSS from the `ps` command:

```
puts "%dM" % `ps -o rss= -p #{Process.pid}`.to_i
```

On Windows your best bet is to use the `OS.rss` function from the OS gem, https://github.com/rdp/os. The gem is outdated and unmaintained, but it still should work for you.

chp1/example_measure_memory.rb
```
require "benchmark"
```

```
num_rows = 100000
num_cols = 10
data = Array.new(num_rows) { Array.new(num_cols) { "x"*1000 } }

puts "%d MB" % (`ps -o rss= -p #{Process.pid}`.to_i/1024)

GC.disable
time = Benchmark.realtime do
  csv = data.map { |row| row.join(",") }.join("\n")
end

puts "%d MB" % (`ps -o rss= -p #{Process.pid}`.to_i/1024)
puts time.round(2)
```

```
$ rbenv shell 2.2.0
$ ruby example_measure_memory.rb
1040 MB
2958 MB
```

Aha. Things are getting more and more interesting. Our initial dataset is roughly 1 gigabyte. Here and later in this book when I write kB I mean 1024 bytes, MB - 1024 * 1024 bytes, GB - 1024 * 1024 * 1024 bytes (yes, I know, it's old school). So, we consumed *2 extra gigabytes* of memory to process that 1 GB of data. Your gut feeling is that it should have taken only 1 GB extra. Instead we took 2 GB. No wonder GC has a lot of work to do!

You probably have a bunch of questions already. Why did the program need 2 GB instead of 1 GB? How do we deal with this? Is there a way for our code to use less memory? The answers are in the next section, but first let's review what we've learned so far.

Takeaways

- Memory consumption and garbage collection are among the major reasons why Ruby is slow.
- Ruby has a significant memory overhead.
- GC in Ruby 2.1 and later is up to five times faster than in earlier versions.
- The raw performance of all modern Ruby interpreters is about the same.