The Life-Changing Magic of Tidying Ruby Object Allocations

16 Sep 2020

Twitter Reddit Lobste.rs Hacker News

Your app is slow. It does not spark joy. This post will show you how to use memory allocation profiling tools to discover performance hotspots, even when they're coming from inside a library. We will use this technique with a real-world application to identify a piece of optimizable code in Active Record that ultimately leads to a patch with a substantial impact on page speed.

> Read the original post on the Heroku engineering blog

In addition to the talk, I've gone back and written a full technical recap of each section to revisit it any time you want without going through the video.

I make heavy use of theatrics here, including a Japanese voiceover artist, animoji, and some edited clips of Marie Kondo's Netflix TV show. This recording was done at EuRuKo on a boat. If you've got the time, here's the talk:

EuRuKo 2019: Tidying Active Record Allocations by Richard Schneeman

Jump to:

Intro to Tidying Object Allocations

The core premise of this talk is that we all want faster applications. Here I'm making the pitch that you can get significant speedups by focusing on your object allocations. To do that, I'll eventually show you a few real-world cases of PRs I made to Rails along with a "how-to" that shows how I used profiling and benchmarking to find and fix the hotspots.

At a high level, the "tidying" technique looks like this:

1. Take all your object allocations and put them in a pile where you can see them
2. Consider each one: Does it spark joy?
3. Keep only the objects that spark joy

An object sparks joy if it is useful, keeps your code clean, and does not cause performance problems. If an object is absolutely necessary, and removing it causes your code to crash, it sparks joy.

To put object allocations in front of us we'll use:

- memory_profiler
- derailed_benchmarks

To get a sense of the cost of object allocation, we can benchmark two different ways to perform the same logic. One of these allocates an array while the other does not.

```ruby
require 'benchmark/ips'

def compare_max(a, b)
  return a if a > b
  b
end

def allocate_max(a, b)
  array = [a, b] # <====== Array allocation here
  array.max
end

Benchmark.ips do |x|
  x.report("allocate_max") {
    allocate_max(1, 2)
  }
  x.report("compare_max ") {
    compare_max(1, 2)
  }
  x.compare!
end
```

This gives us the results:

```
Warming up --------------------------------------
        allocate_max   258.788k i/100ms
        compare_max    307.196k i/100ms
Calculating -------------------------------------
        allocate_max      6.665M (±14.6%) i/s -     32.090M in   5.033786s
        compare_max      13.597M (± 6.0%) i/s -     67.890M in   5.011819s

Comparison:
        compare_max : 13596747.2 i/s
        allocate_max:  6664605.5 i/s - 2.04x  slower
```

In this example, allocating an array is 2x slower than making a direct comparison. It's a truism in most languages that allocating memory or creating objects is slow. In the `C` programming language, it's a truism that "malloc is slow."

Since we know that allocating in Ruby is slow, we can make our programs faster by removing allocations. As a simplifying assumption, I've found that a decrease in bytes allocated roughly corresponds to performance improvement. For example, if I can reduce the number of bytes allocated by 1% in a request, then on average, the request will have been sped up by about 1%. This assumption helps us benchmark faster as it's much easier to measure memory allocated than it is to repeatedly run hundreds or thousands of timing benchmarks.

Tidying Example 1: Active Record `respond_to?` logic

Using the target application CodeTriage.com and derailed benchmarks, we get a "pile" of memory allocations:

```
$ bundle exec derailed exec perf:objects

allocated memory by gem
-----------------------------------
    227058  activesupport/lib
    134366  codetriage/app
    # ...


allocated memory by file
-----------------------------------
    126489  …/code/rails/activesupport/lib/active_support/core_ext/string/ou
     49448  …/code/codetriage/app/views/layouts/_app.html.slim
     49328  …/code/codetriage/app/views/layouts/application.html.slim
     36097  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
     25096  …/code/codetriage/app/views/pages/_repos_with_pagination.html.sl
     24432  …/code/rails/activesupport/lib/active_support/core_ext/object/to
     23526  …/code/codetriage/.gem/ruby/2.5.3/gems/rack-mini-profiler-1.0.0/
     21912  …/code/rails/activerecord/lib/active_record/connection_adapters/
     18000  …/code/rails/activemodel/lib/active_model/attribute_set/builder.
     15888  …/code/rails/activerecord/lib/active_record/result.rb
     14610  …/code/rails/activesupport/lib/active_support/cache.rb
```

```
      11109  …/code/codetriage/.gem/ruby/2.5.3/gems/rack-mini-profiler-1.0.0/
       9824  …/code/rails/actionpack/lib/abstract_controller/caching/fragment
       9360  …/.rubies/ruby-2.5.3/lib/ruby/2.5.0/logger.rb
       8440  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
       8304  …/code/rails/activemodel/lib/active_model/attribute.rb
       8160  …/code/rails/actionview/lib/action_view/renderer/partial_rendere
       8000  …/code/rails/activerecord/lib/active_record/integration.rb
       7880  …/code/rails/actionview/lib/action_view/log_subscriber.rb
       7478  …/code/rails/actionview/lib/action_view/helpers/tag_helper.rb
       7096  …/code/rails/actionview/lib/action_view/renderer/partial_rendere
       # ...
```

The full output is massive, so I've truncated it here.

Once you've got your memory in a pile. I like to look at the "allocated memory" by file. I start at the top and look at each in turn. In this case, we'll look at this file:

```
       8440  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
```

Once you have a file you want to look at, you can focus on it in derailed like this:

```
$ ALLOW_FILES=active_record/attribute_methods.rb \
  bundle exec derailed exec perf:objects

allocated memory by file
-----------------------------------
      8440  …/code/rails/activerecord/lib/active_record/attribute_methods.rb

allocated memory by location
-----------------------------------
      8000  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
       320  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
        80  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
        40  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
```

Now we can see exactly where the memory is being allocated in this file. Starting at the top of the locations, I'll work my way down to understand how memory is allocated and used.

Looking first at this line:

```
        8000  …/code/rails/activerecord/lib/active_record/attribute_methods.rb
```

We can open this in an editor and navigate to that location:

```
$ bundle open activerecord
```

In that file, here’s the line allocating the most memory:

```
def respond_to?(name, include_private = false)
  return false unless super

  case name
  when :to_partial_path
    name = "to_partial_path"
  when :to_model
    name = "to_model"
  else
    name = name.to_s # <=== Line 270 here
  end

  # If the result is true then check for the select case.
  # For queries selecting a subset of columns, return false for unselec
  # We check defined?(@attributes) not to issue warnings if called on o
  # have been allocated but not yet initialized.
  if defined?(@attributes) && self.class.column_names.include?(name)
    return has_attribute?(name)
  end

  true
end
```

Here we can see on line 270 that it's allocating a string. But why? To answer that question, we need more context. We need to understand how this code is used. When we call `respond_to` on an object, we want to know if a method by that name exists. Because Active Record is backed by a database, it needs to see if a column exists with that name.

Typically when you call `respond_to` you pass in a symbol, for example, `user.respond_to?(:email)`. But in Active Record, columns are stored as strings. On line 270, we're ensuring that the `name` value is always a string.

This is the code where name is used:

```ruby
    if defined?(@attributes) && self.class.column_names.include?(name)
```

Here `column_names` returns an array of column names, and the `include?` method will iterate over each until it finds the column with that name, or its nothing (`nil`).

To determine if we can get rid of this allocation, we have to figure out if there's a way to replace it without allocating memory. We need to refactor this code while maintaining correctness. I decided to add a method that converted the array of column names into a hash with symbol keys and string values:

```ruby
# lib/activerecord/model_schema.rb
def symbol_column_to_string(name_symbol) # :nodoc:
  @symbol_column_to_string_name_hash ||= column_names.index_by(&:to_sym
  @symbol_column_to_string_name_hash[name_symbol]
end
```

This is how you would use it:

```ruby
User.symbol_column_to_string(:email) #=> "email"
User.symbol_column_to_string(:foo)   #=> nil
```

Since the value that is being returned every time by this method is from the same hash, we can re-use the same string and not have to allocate. The refactored `respond_to` code ends up looking like this:

```
def respond_to?(name, include_private = false)
  return false unless super

  # If the result is true then check for the select case.
  # For queries selecting a subset of columns, return false for unselec
  # We check defined?(@attributes) not to issue warnings if called on o
  # have been allocated but not yet initialized.
  if defined?(@attributes)
    if name = self.class.symbol_column_to_string(name.to_sym)
      return has_attribute?(name)
    end
  end

  true
end
```

Running our benchmarks, this patch yielded a reduction in memory of 1%. Using code that eventually became `derailed exec perf:library`, I verified that the patch made end-to-end request/response page speed on CodeTriage 1% faster.

Performance and Statistical Significance

When talking about benchmarks, it's important to talk about statistics and their impact. I talk a bit about this in Lies, Damned Lies, and Averages: Perc50, Perc95 explained for Programmers. Essentially any time you measure a value, there's a chance that it could result from randomness. If you run a benchmark 3 times, it will give you 3 different results. If it shows that it was faster twice and slower once, how can you be certain that the results are because of the change and not random chance?

That’s precisely the question that “statistical significance” tries to answer. While we can never know, we can make an informed decision. How? Well, if you took a measurement of the same code many times, you would know any variation was the result of randomness. This would give you a distribution of randomness. Then you could use this distribution to understand how likely it is that your change was caused by randomness.

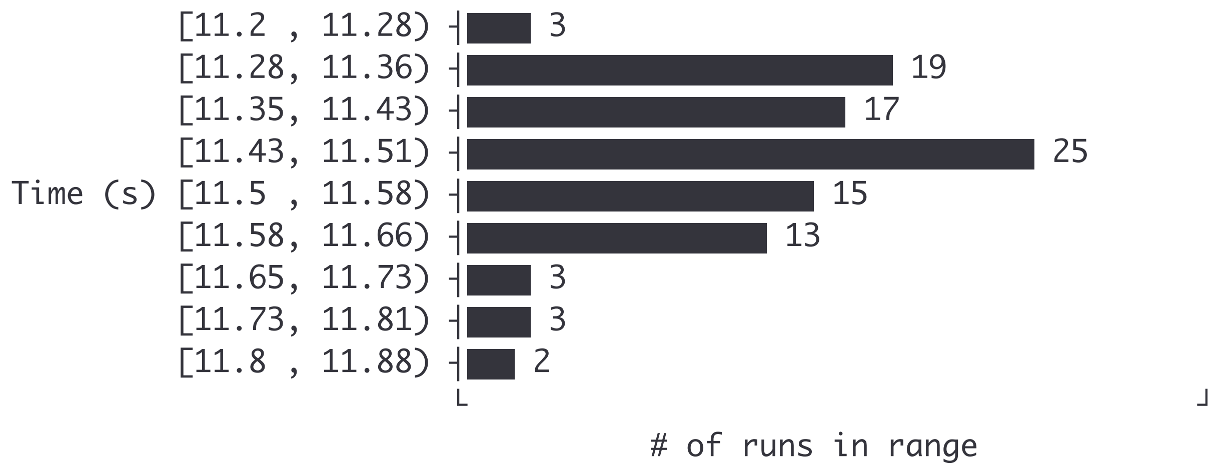
In the talk, I go into detail on the origins of “Student’s T-Test.” In derailed, I’ve switched to using Kolmogorov-Smirnov instead. When I ran benchmarks on CodeTriage, I wanted to be sure that my results were valid, so I ran them multiple times and ran Kolmogorov Smirnov on them. This gives me a confidence interval. If my results are in that interval, then I can say with 95% certainty that my results are not the result of random chance i.e., that they’re valid and are statistically significant.

If it’s not significant, it could mean that the change is too small to detect, that you need more samples, or that there is no difference.

In addition to running a significance check on your change, it’s useful to see the distribution. Derailed benchmarks does this for you by default now. Here is a result from `derailed exec perf:library` used to compare the performance difference of two different commits in a library dependency:

```
                  Histogram - [winner] "I am the new commit."
                                ┌                               ┐
             [11.2 , 11.28) ┤▇▇▇▇▇▇▇▇▇ 12
             [11.28, 11.36) ┤▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 22
             [11.35, 11.43) ┤▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 30
             [11.43, 11.51) ┤▇▇▇▇▇▇▇▇▇▇▇▇▇ 17
    Time (s) [11.5 , 11.58) ┤▇▇▇▇▇▇▇▇▇▇ 13
             [11.58, 11.66) ┤▇▇▇▇▇ 6
             [11.65, 11.73) ┤ 0
             [11.73, 11.81) ┤ 0
             [11.8 , 11.88) ┤ 0
                                └                               ┘
                                        # of runs in range


                  Histogram - [loser] "Old commit"
                                ┌                               ┐
```



The TLDR of this whole section is that in addition to showing my change as being faster, I was also able to show that the improvement was statistically significant.

Tidying example 2: Converting strings to time takes time

One percent faster is good, but it could be better. Let’s do it again. First, get a pile of objects:

```
$ bundle exec derailed exec perf:objects

# ...

allocated memory by file
-----------------------------------
    126489  …/code/rails/activesupport/lib/active_support/core_ext/string/ou
     49448  …/code/codetriage/app/views/layouts/_app.html.slim
     49328  …/code/codetriage/app/views/layouts/application.html.slim
     36097  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
     25096  …/code/codetriage/app/views/pages/_repos_with_pagination.html.sl
     24432  …/code/rails/activesupport/lib/active_support/core_ext/object/to
     23526  …/code/codetriage/.gem/ruby/2.5.3/gems/rack-mini-profiler-1.0.0/
     21912  …/code/rails/activerecord/lib/active_record/connection_adapters/
     18000  …/code/rails/activemodel/lib/active_model/attribute_set/builder.
     15888  …/code/rails/activerecord/lib/active_record/result.rb
     14610  …/code/rails/activesupport/lib/active_support/cache.rb
     11148  …/code/codetriage/.gem/ruby/2.5.3/gems/rack-mini-profiler-1.0.0/
      9824  …/code/rails/actionpack/lib/abstract_controller/caching/fragment
      9360  …/.rubies/ruby-2.5.3/lib/ruby/2.5.0/logger.rb
      8304  …/code/rails/activemodel/lib/active_model/attribute.rb
```

Zoom in on a file:

```
     36097  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
```

Isolate the file:

```
$ ALLOW_FILE=active_model/type/helpers/time_value.rb \
  bundle exec derailed exec perf:objects

Total allocated: 39617 bytes (600 objects)
Total retained:  0 bytes (0 objects)

allocated memory by gem
-----------------------------------
     39617  activemodel/lib

allocated memory by file
-----------------------------------
     39617  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu

allocated memory by location
-----------------------------------
     17317  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
     12000  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
      6000  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
      4300  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
```

We’re going to do the same thing by starting to look at the top location:

```
     17317  …/code/rails/activemodel/lib/active_model/type/helpers/time_valu
```

Here’s the code:

```ruby
def fast_string_to_time(string)
 if string =~ ISO_DATETIME # <=== line 72 Here
   microsec = ($7.to_r * 1_000_000).to_i
   new_time $1.to_i, $2.to_i, $3.to_i, $4.to_i, $5.to_i, $6.to_i, micro
 end
end
```

On line 72, we are matching the input string with a regular expression constant. This allocates a lot of memory because each grouped match of the regular expression allocates a new string. To understand if we can make this faster, we have to understand how it's used.

This method takes in a string, then uses a regex to split it into parts, and then sends those parts to the `new_time` method.

There's not much going on that can be sped up there, but what's happening on this line:

```ruby
   microsec = ($7.to_r * 1_000_000).to_i
```

Here's the regex:

```ruby
ISO_DATETIME = /\A(\d{4})-(\d\d)-(\d\d) (\d\d):(\d\d):(\d\d)(\.\d+)?\z/
```

When I ran the code and output $7 from the regex match, I found that it would contain a string that starts with a dot and then has numbers, for example:

```ruby
puts $7 # => ".1234567"
```

This code wants microseconds as an integer, so it turns it into a “rational” and then multiplies it by a million and turns it into an integer.

```
($7.to_r * 1_000_000).to_i # => 1234567
```

You might notice that it looks like we’re basically dropping the period and then turning it into an integer. So why not do that directly?

Here’s what it looks like:

```
def fast_string_to_time(string)
  if string =~ ISO_DATETIME
    microsec_part = $7
    if microsec_part && microsec_part.start_with?(".") && microsec_part
      microsec_part[0] = ""          # <=== HERE
      microsec = microsec_part.to_i # <=== HERE
    else
      microsec = (microsec_part.to_r * 1_000_000).to_i
    end
    new_time $1.to_i, $2.to_i, $3.to_i, $4.to_i, $5.to_i, $6.to_i, micr
  end
```

We’ve got to guard this case by checking for the conditions of our optimization. Now the question is: is this faster?

Here’s a microbenchmark:

```
original_string = ".443959"

require 'benchmark/ips'

Benchmark.ips do |x|
  x.report("multiply") {
      string = original_string.dup
      (string.to_r * 1_000_000).to_i
    }
    x.report("new      ") {
      string = original_string.dup
      if string && string.start_with?(".".freeze) && string.length == 7
        string[0] = ''.freeze
        string.to_i
      end
    }
    x.compare!
  end

  # Warming up --------------------------------------
  #             multiply   125.783k i/100ms
  #             new        146.543k i/100ms
  # Calculating -------------------------------------
  #             multiply      1.751M (± 3.3%) i/s -      8.805M in   5.03
  #             new           2.225M (± 2.1%) i/s -     11.137M in   5.00

  # Comparison:
  #             new      :  2225289.7 i/s
  #             multiply:  1751254.2 i/s - 1.27x  slower
```

The original code is 1.27x slower. YAY!

Tidying Example 3: Lightning fast cache keys

The last speedup is kind of underwhelming, so you might wonder why I added it. If you remember our first example of optimizing `respond_to`, it helped to understand the broader context of how it's used. Since this is such an expensive object allocation location, is there an opportunity to call it less or not call it at all?

To find out, I added a `puts caller` in the code and re-ran it. Here's part of a backtrace:

```
==========================================================================
…/code/rails/activemodel/lib/active_model/type/date_time.rb:25:in `cast_valu
…/code/rails/activerecord/lib/active_record/connection_adapters/postgresql/c
…/code/rails/activemodel/lib/active_model/type/value.rb:38:in `cast'
…/code/rails/activemodel/lib/active_model/type/helpers/accepts_multiparamete
…/code/rails/activemodel/lib/active_model/type/value.rb:24:in `deserialize'
…/.rubies/ruby-2.5.3/lib/ruby/2.5.0/delegate.rb:349:in `block in delegating_
…/code/rails/activerecord/lib/active_record/attribute_methods/time_zone_conv
…/code/rails/activemodel/lib/active_model/attribute.rb:164:in `type_cast'
…/code/rails/activemodel/lib/active_model/attribute.rb:42:in `value'
…/code/rails/activemodel/lib/active_model/attribute_set.rb:48:in `fetch_valu
…/code/rails/activerecord/lib/active_record/attribute_methods/read.rb:77:in
…/code/rails/activerecord/lib/active_record/attribute_methods/read.rb:40:in
…/code/rails/activesupport/lib/active_support/core_ext/object/try.rb:16:in `
…/code/rails/activesupport/lib/active_support/core_ext/object/try.rb:16:in `
…/code/rails/activerecord/lib/active_record/integration.rb:99:in `cache_vers
…/code/rails/activerecord/lib/active_record/integration.rb:68:in `cache_key'
…/code/rails/activesupport/lib/active_support/cache.rb:639:in `expanded_key'
…/code/rails/activesupport/lib/active_support/cache.rb:644:in `block in expa
…/code/rails/activesupport/lib/active_support/cache.rb:644:in `collect'
…/code/rails/activesupport/lib/active_support/cache.rb:644:in `expanded_key'
…/code/rails/activesupport/lib/active_support/cache.rb:608:in `normalize_key
…/code/rails/activesupport/lib/active_support/cache.rb:565:in `block in read
…/code/rails/activesupport/lib/active_support/cache.rb:564:in `each'
…/code/rails/activesupport/lib/active_support/cache.rb:564:in `read_multi_en
…/code/rails/activesupport/lib/active_support/cache.rb:387:in `block in read
```

I followed it backwards until I hit these two places:

```
…/code/rails/activerecord/lib/active_record/integration.rb:99:in `cache_vers
…/code/rails/activerecord/lib/active_record/integration.rb:68:in `cache_key'
```

It looks like this expensive code is being called while generating a cache key.

```
def cache_key(*timestamp_names)
  if new_record?
    "#{model_name.cache_key}/new"
  else
    if cache_version && timestamp_names.none? # <== line 68 here
```

```
      "#{model_name.cache_key}/#{id}"
    else
      timestamp = if timestamp_names.any?
        ActiveSupport::Deprecation.warn(<<-MSG.squish)
          Specifying a timestamp name for #cache_key has been deprecate
          the explicit #cache_version method that can be overwritten.
        MSG

        max_updated_column_timestamp(timestamp_names)
      else
        max_updated_column_timestamp
      end

      if timestamp
        timestamp = timestamp.utc.to_s(cache_timestamp_format)
        "#{model_name.cache_key}/#{id}-#{timestamp}"
      else
        "#{model_name.cache_key}/#{id}"
      end
    end
  end
end
```

On line 68 in the `cache_key` code it calls `cache_version`. Here's the code for `cache_version`:

```
def cache_version # <== line 99 here
  if cache_versioning && timestamp = try(:updated_at)
    timestamp.utc.to_s(:usec)
  end
end
```

Here is our culprit:

```
timestamp = try(:updated_at)
```

What is happening is that some database adapters, such as the one for Postgres, returned their values from the database driver as strings. Then active record will lazily cast them into Ruby objects when they are needed. In this case, our time value method is being called to convert the updated timestamp into a time object so we can use it to generate a cache version string.

Here's the value before it's converted:

```
User.first.updated_at_before_type_cast # => "2019-04-24 21:21:09.232249
```

And here's the value after it's converted:

```
User.first.updated_at.to_s(:usec)      # => "20190424212109232249"
```

Basically, all the code is doing is trimming out the non-integer characters. Like before, we need a guard that our optimization can be applied:

```
# Detects if the value before type cast
# can be used to generate a cache_version.
#
# The fast cache version only works with a
# string value directly from the database.
#
# We also must check if the timestamp format has been changed
# or if the timezone is not set to UTC then
# we cannot apply our transformations correctly.
def can_use_fast_cache_version?(timestamp)
  timestamp.is_a?(String) &&
    cache_timestamp_format == :usec &&
    default_timezone == :utc &&
    !updated_at_came_from_user?
end
```

Then once we're in that state, we can modify the string directly:

```
# Converts a raw database string to `:usec`
# format.
#
# Example:
#
#   timestamp = "2018-10-15 20:02:15.266505"
#   raw_timestamp_to_cache_version(timestamp)
#   # => "20181015200215266505"
#
# PostgreSQL truncates trailing zeros,
# https://github.com/postgres/postgres/commit/3e1beda2cde3495f41290e1ec
# to account for this we pad the output with zeros
def raw_timestamp_to_cache_version(timestamp)
  key = timestamp.delete("- :.")
  if key.length < 20
    key.ljust(20, "0")
  else
    key
  end
end
```

There's some extra logic due to the Postgres truncation behavior linked above. The resulting code to `cache_version` becomes:

```
def cache_version
  return unless cache_versioning

  if has_attribute?("updated_at")
    timestamp = updated_at_before_type_cast
    if can_use_fast_cache_version?(timestamp)
```

```
        raw_timestamp_to_cache_version(timestamp)
      elsif timestamp = updated_at
        timestamp.utc.to_s(cache_timestamp_format)
      end
    end
  end
```

That’s the opportunity. What’s the impact?

```
Before: Total allocated: 743842 bytes (6626 objects)
After:  Total allocated: 702955 bytes (6063 objects)
```

The bytes reduced is 5% fewer allocations. Which is pretty good. How does it translate to speed?

It turns out that time conversion is very CPU intensive and changing this code makes the target application up to 1.12x faster. This means that if your app previously required 100 servers to run, it can now run with about 88 servers.

Wrap up

Adding together these optimizations and others brings the overall performance improvement to 1.23x or a net reduction of 19 servers. Basically, it’s like buying 4 servers and getting 1 for free.

These examples were picked from my changes to the Rails codebase, but you can use them to optimize your applications. The general framework looks like this:

- Get a list of all your memory
- Zoom in on a hotspot
- Figure out what is causing that memory to be allocated inside of your code
- Ask if you can refactor your code to not depend on those allocations

If you want to learn more about memory, here are my recommendations:

- Why does my App's Memory Use Grow Over Time? - Start here, an excellent high-level overview of what causes a system's memory to grow that will help you develop an understanding of how Ruby allocates and uses memory at the application level.
- Complete Guide to Rails Performance (Book) - This book is by Nate Berkopec and is excellent. I recommend it to someone at least once a week.
- How Ruby uses memory - This is a lower level look at precisely what "retained" and "allocated" memory means. It uses small scripts to demonstrate Ruby memory behavior. It also explains why the "total max" memory of our system rarely goes down.
- How Ruby uses memory (Video) - If you're new to the concepts of object allocation, this might be an excellent place to start (you can skip the first story in the video, the rest are about memory). Memory stuff starts at 13 minutes
- Jumping off the Ruby Memory Cliff - Sometimes you might see a 'cliff' in your memory metrics or a saw-tooth pattern. This article explores why that behavior exists and what it means.
- Ruby Memory Use (Heroku Devcenter article I maintain) - This article focuses on alleviating the symptoms of high memory use.
- Debugging a memory leak on Heroku - TLDR; It's probably not a leak. Still worth reading to see how you can come to the same conclusions yourself. Content is valid for other environments than Heroku. Lots of examples of using the tool `derailed_benchmarks` (that I wrote).
- The Life-Changing Magic of Tidying Active Record Allocations (Post + Video) - This talk shows how I used tools to track down and eliminate memory allocations in real life. All of the examples are from patches I submitted to Rails, but the process works the same for finding allocations caused by your application logic.

Get ahold of Richard and stay up-to-date with Ruby, Rails, and other programming related content through a subscription to his mailing list.