Variable Width Allocation: Strings

GitHub PR: https://github.com/ruby/ruby/issues/4933

Feature description

Since merging #18045 which introduced size pools inside the GC to allocate various sized slots, we've been working on expanding the usage of VWA into more types (before this patch, only classes are allocated through VWA). This patch changes strings to use VWA.

Summary

- This patch allocates strings using VWA.
- String allocated through VWA are embedded. The patch changes strings to support dynamic capacity embedded strings.
- We do not handle resizing in VWA in this patch (embedded strings resized up are moved back to malloc), which may result in wasted space. However, in benchmarks, this does not appear to be an issue.
- We propose enabling VWA by default. We are confident about the stability and performance of this feature.

String allocation

Strings with known sizes at allocation time that are small enough is allocated as an embedded string. Embedded string headers are now 18 bytes, so the maximum embedded string length is now 302 bytes (currently VWA is configured to allocate up to 320 byte slots, but can be easily configured for even larger slots). Embedded strings have their contents directly follow the object headers. This aims to improve cache performance since the contents are on the same cache line as the object headers. For strings with unknown sizes, or with contents that are too large, it falls back to allocate 40 byte slots and store the contents in the malloc heap.

String reallocation

If an embedded string is expanded and can no longer fill the slot, it is moved into the malloc heap. This may mean that some space in the slot is wasted. For example, if the string was originally allocated in a 160 byte slot and moved to the malloc heap, 120 bytes of the slot is wasted (since we only need 40 bytes for a string with contents on the malloc heap). This patch does not aim to tackle this issue. Memory usage also does not appear to be an issue in any of the benchmarks.

Incompatibility of VWA and non-VWA built gems

Gems with native extensions built with VWA are not compatible with non-VWA built gems (and vice versa). When switching between VWA and non-VWA rubies, gems must be rebuilt. This is because the header file rstring.h changes depending on whether USE_RVARGC is set or not (the internal string implementation changes with VWA).

Enabling VWA by default

In this patch, we propose enabling VWA by default. We believe that the performance data supports this (see the benchmark results section). We're also confident about its stability. It passess all tests on CI for the Shopify monolith and we've ran this version of Ruby on a small portion of production traffic in a Shopify service for about a week (where it served over 500 million requests).

Although VWA is enabled by default, we plan on supporting the USE_RVARGC flag as an escape hatch to disable VWA until at least Ruby 3.1 is released (i.e. we may remove it starting 2022).

Benchmark setup

Benchmarking was done on a bare-metal Ubuntu machine on AWS. All benchmark results are using glibc by default, except when
jemalloc is explicitly specified.

```
$ uname -a
Linux 5.8.0-1038-aws #40~20.04.1-Ubuntu SMP Thu Jun 17 13:25:28 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
```

glibc version:

```
$ ldd --version
ldd (Ubuntu GLIBC 2.31-0ubuntu9.2) 2.31
```

To measure memory usage over time, the mstat tool was used.

Ruby master was benchmarked on commit 7adfb14f60. The branch was rebased on top of the same commit.

Performance benchmarks of this branch without VWA turned on are included to make sure this patch does not introduce a performance regression compared to master.

### Benchmark results

#### Summary

- On a live, production Shopify service, we see no significant differences in response times. However, VWA has lower memory usage.
- On railsbench, we see a minor performance improvement. However, we also see worse p100 response times. The reason is analyzed in the railsbench section below. VWA uses more memory when using glibc, but uses an equal amount of memory for jemalloc.
- On rdoc generation, VWA uses significantly less memory for a small reduction in performance.
- Microbenchmarks show significant performance improvement for strings that are embedded in VWA (e.g. String#== is significantly faster).

#### Shopify production

We deployed this branch with VWA enabled vs. Ruby master commit 7adfb14f60 on a small portion of live, production traffic for a Shopify web application to test real-world performance. This data was collected for a period of 1 day where they each served approximately 4 million requests. Average, median, p90, p99 response times were within the margin of error of each other (<1% difference). However, VWA consistently had lower memory usage (Resident Set Size) by 0.96x.

#### railsbench

For railsbench, we ran the railsbench benchmark. For both the performance and memory benchmarks, 25 runs were conducted for each combination (branch + glibc, master + glibc, branch + jemalloc, master + jemalloc).

In this benchmark, VWA suffers from poor p100. This is because railsbench is relatively small (only one controller and model), so there are not many pages in each size pools. Incremental marking requires there to be many pooled pages, but because there are not a lot of pooled pages (due to the small heap size), it has to mark a very large number of objects at every step. We do not expect this to be a problem for real apps with a larger number of objects allocated at boot, and this is confirmed by metrics collected in the Shopify production application.

#### glibc

Using glibc, VWA is about 1.018x faster than master in railsbench in throughput (RPS).
<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS</td>
<td>740.92</td>
<td>722.97</td>
<td>745.31</td>
</tr>
<tr>
<td>p50 (ms)</td>
<td>1.31</td>
<td>1.37</td>
<td>1.33</td>
</tr>
<tr>
<td>p90 (ms)</td>
<td>1.40</td>
<td>1.45</td>
<td>1.43</td>
</tr>
<tr>
<td>p99 (ms)</td>
<td>2.36</td>
<td>1.80</td>
<td>1.78</td>
</tr>
<tr>
<td>p100 (ms)</td>
<td>21.38</td>
<td>17.28</td>
<td>15.15</td>
</tr>
</tbody>
</table>

Average max memory usage for VWA: 104.82 MB
Average max memory usage for master: 101.17 MB
VWA uses 1.04x more memory.

jemalloc

Using glibc, VWA is about 1.06x faster than master in railsbench in RPS.

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS</td>
<td>781.21</td>
<td>742.12</td>
<td>739.04</td>
</tr>
<tr>
<td>p50 (ms)</td>
<td>1.25</td>
<td>1.31</td>
<td>1.30</td>
</tr>
<tr>
<td>p90 (ms)</td>
<td>1.32</td>
<td>1.39</td>
<td>1.38</td>
</tr>
<tr>
<td>p99 (ms)</td>
<td>2.15</td>
<td>2.53</td>
<td>4.41</td>
</tr>
<tr>
<td>p100 (ms)</td>
<td>21.11</td>
<td>16.60</td>
<td>15.75</td>
</tr>
</tbody>
</table>

Average max memory usage for VWA: 102.68 MB
Average max memory usage for master: 103.14 MB
VWA uses 1.00x less memory.

rdoc generation

In rdoc generation, we see significant memory usage reduction at the cost of small performance reduction.

glibc

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
<th>VWA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>16.68</td>
<td>16.08</td>
<td>15.87</td>
<td>0.95x</td>
</tr>
</tbody>
</table>

Average max memory usage for VWA: 295.19 MB
Average max memory usage for master: 365.06 MB
VWA uses 0.81x less memory.

jemalloc

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
<th>VWA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>16.34</td>
<td>15.64</td>
<td>15.51</td>
<td>0.95x</td>
</tr>
</tbody>
</table>

Average max memory usage for VWA: 104.82 MB
Average max memory usage for master: 101.17 MB
VWA uses 1.04x more memory.

jemalloc

Using glibc, VWA is about 1.06x faster than master in railsbench in RPS.

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS</td>
<td>781.21</td>
<td>742.12</td>
<td>739.04</td>
</tr>
<tr>
<td>p50 (ms)</td>
<td>1.25</td>
<td>1.31</td>
<td>1.30</td>
</tr>
<tr>
<td>p90 (ms)</td>
<td>1.32</td>
<td>1.39</td>
<td>1.38</td>
</tr>
<tr>
<td>p99 (ms)</td>
<td>2.15</td>
<td>2.53</td>
<td>4.41</td>
</tr>
<tr>
<td>p100 (ms)</td>
<td>21.11</td>
<td>16.60</td>
<td>15.75</td>
</tr>
</tbody>
</table>

Average max memory usage for VWA: 102.68 MB
Average max memory usage for master: 103.14 MB
VWA uses 1.00x less memory.

rdoc generation

In rdoc generation, we see significant memory usage reduction at the cost of small performance reduction.

glibc

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
<th>VWA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>16.68</td>
<td>16.08</td>
<td>15.87</td>
<td>0.95x</td>
</tr>
</tbody>
</table>

Average max memory usage for VWA: 295.19 MB
Average max memory usage for master: 365.06 MB
VWA uses 0.81x less memory.

jemalloc
Average max memory usage for VWA: 281.16 MB
Average max memory usage for master: 316.49 MB
VWA uses 0.89x less memory.

**Liquid benchmarks**
For the liquid benchmarks, we ran the liquid benchmark averaged over 5 runs each. We see that VWA is faster across the board.

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
<th>VWA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse (i/s)</td>
<td>40.40</td>
<td>38.44</td>
<td>39.47</td>
<td>1.02x</td>
</tr>
<tr>
<td>Render (i/s)</td>
<td>126.47</td>
<td>121.97</td>
<td>121.20</td>
<td>1.04x</td>
</tr>
<tr>
<td>Parse &amp; Render (i/s)</td>
<td>28.81</td>
<td>27.52</td>
<td>28.02</td>
<td>1.03x</td>
</tr>
</tbody>
</table>

**Microbenchmarks**
These microbenchmarks are very favourable for VWA since the strings created have a length of 71, so they are embedded in VWA and allocated on the malloc heap for master.

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
<th>VWA speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>String#== (i/s)</td>
<td>1.806k</td>
<td>1.334k</td>
<td>1.337k</td>
<td>1.35x</td>
</tr>
<tr>
<td>String#times (i/s)</td>
<td>6.010M</td>
<td>5.238M</td>
<td>5.247M</td>
<td>1.15x</td>
</tr>
<tr>
<td>String#[[]=(i/s)</td>
<td>1.031k</td>
<td>863.816</td>
<td>897.915</td>
<td>1.15x</td>
</tr>
</tbody>
</table>

**Benchmark source code**
```ruby
require "bundler/inline"
gemfile do
  source "https://rubygems.org"
  gem "benchmark-ips"
end
COUNT = 10_000
strs1 = []
strs2 = []
nine = "9"
COUNT.times do
  strs1 << ["A".."Z", "0".."9"].join(" ")
  strs2 << ["A".."Z", "0".."9"].join(" ")
end
Benchmark.ips do |x|
  x.report("String#==") do |times|
    i = 0
    while i < times
      COUNT.times { |i| strs1[i] == strs2[i] }
      i += 1
    end
  end
  x.report("String#times") do |times|
    i = 0
    while i < times
      "a" * 100
      i += 1
    end
end
```
```ruby
end

x.report("String\[\]=") do |times|
  i = 0
  while i < times
    strs1.each { |str| str[-1] = nine }
    i += 1
  end
end

Associated revisions
Revision 46b6e6eb9 - 10/25/2021 05:26 PM - peterzhu2118 (Peter Zhu)
[Feature #18239] Add struct for embedded strings

Revision 6374be5a - 10/25/2021 05:26 PM - peterzhu2118 (Peter Zhu)
[Feature #18239] Refactor RVARGC alloc functions
The allocation functions no longer assume that one RVALUE needs to be allocated.

Revision a5b65981 - 10/25/2021 05:26 PM - peterzhu2118 (Peter Zhu)
[Feature #18239] Implement VWA for strings
This commit adds support for embedded strings with variable capacity and uses Variable Width Allocation to allocate strings.

History
#1 - 10/04/2021 06:29 PM - ko1 (Koichi Sasada)
Thank you for your work. I have several questions.

- Could you explain the RString layout?
- Could you explain the variable size page strategy? https://bugs.ruby-lang.org/issues/18045 shows 4 sizes (40, 80, 160, 320). Same with it? For example, jemalloc employs more fine-grained pools. Did you try other sizes?
- Did you measure the allocation overhead on non-main ractors?

I didn't ask details because I didn't think it was enabled on 3.1.

#2 - 10/04/2021 06:32 PM - ko1 (Koichi Sasada)
ko1 (Koichi Sasada) wrote in #note-1:

- Could you explain the variable size page strategy? https://bugs.ruby-lang.org/issues/18045 shows 4 sizes (40, 80, 160, 320). Same with it? For example, jemalloc employs more fine-grained pools. Did you try other sizes?

To find out good tuning, maybe allocation size measurement on apps will help.

Another question:

- Does it support current GC features like compaction and incremental marking?

#3 - 10/04/2021 06:49 PM - peterzhu2118 (Peter Zhu)
Hi ko1 (Koichi Sasada), thank you for the feedback.

- The RString layout does not change much. The only big change is that the length for embedded strings is moved from using 5 bits in the header to a field in the RString. This is because the 5 bits limits the length of a maximum 32 bytes.
- The slot sizes have not changed (40B, 80B, 160B, 320B). We have not tried other sizes. Memory usage was not a problem in any of the benchmarks we ran (in most cases VWA had lower memory usage than master). In the future, we plan on switching to powers-of-2 sizes, so we may revisit this issue and try more sizes.
- For non-40 byte size pools, we do not currently optimize for ractors (allocation requires locking the VM). Implementing this will be one of our next things to do.
- VWA fully supports compaction and incremental marking. For compaction, objects are only compacted within each size pool (e.g. a 160B object is only compacted to another 160B slot). In the future, we plan on investigating compaction across different size pools.

03/05/2022
Sorry for incremental questions.

Why do we need to enable it now? Current benchmark you showed illustrate not so difference.

"On a live, production Shopify service, we see no significant differences in response times. However, VWA has lower memory usage." is the motivation? (and other rails app can be expect similar?)

It can improve the performance if other types utilize VWA.

Can we wait for it or not?

In general, current implementation (Fixed-width allocation: FWA) is middle position. My understanding:

- **Memory usage**
  - M1. VWA can be better than FWA because the malloc header is not needed.
  - M2. VWA can be worse than FWA because the waste size problem
    - M2-1. sized-page issue: if there are many empty slots in size-X, but size-Y is full, we can't utilize them.
    - M2-2. if size 161 is needed and 320B slot is allocated (159B is wasted).
    - M2-3. re-size issue described in this issue.

  For M2-1/M2-3, compaction can fix.
  For M2-2, the page size is important (for example, 320B can be too big).

- **Performance**
  - P1. VWA can be better than FWA because of cache locality (the RVALUE and the contents can be nearer than FWA).
  - P2. Allocation time of VWA can be slower than FWA to select the pool ... but is it a matter?
  - P3. GC (sweeping) can be slower than FWA because of M2-1.

I can't make good reason why VWA is slower than FWA on your benchmark.

I agree "Let's challenge new implementation if it doesn't have big drawback" is one idea.
I want to clear the reason.

---

The RString layout does not change much. The only big change is that the length for embedded strings is moved from using 5 bits in the header to a field in the RString. This is because the 5 bits limits the length of a maximum 32 bytes.

So flags doesn't have a size and every String has 1 word (8B on 64bit CPU) length field?

https://github.com/ruby/ruby/pull/4933/files#diff-6d6e2cbec5870ad781a30b7610a631a2b88566961bd8324ba4c08b2b675dc13dR288

Ah, I see. It is "embed" and short length field + contents field (char[]).

- The slot sizes have not changed (40B, 80B, 160B, 320B). We have not tried other sizes. Memory usage was not a problem in any of the benchmarks we ran (in most cases VWA had lower memory usage than master). In the future, we plan on switching to powers-of-2 sizes, so we may revisit this issue and try more sizes.

Is it easy to increase/decrease the sizes? or optimized for 4 sizes?

**Performance:**

- P4. GC.compact can be slower than FWA because it should be copy the body. But maybe it is not a matter.

I can't make good reason why VWA is slower than FWA on your benchmark.

The only benchmark above where VWA is slower than FWA is rdoc generation. The performance decrease is small but the memory decrease is large. This is because VWA can be tighter on the number of heap pages for size pools where there is not a lot of allocations after boot (e.g. 160B size pools for classes). This sometimes results in requiring more GC runs.

On railsbench, VWA is faster on RPS (requests per second). It has worse p100 response times because of the smaller heap sizes of each size pool, so it does less incremental marking steps than FWA.
Is it easy to increase/decrease the sizes? or optimized for 4 sizes?

It's very easy to change the number of size pools. We chose 4 because it's not too small and not too large. Any number between 1 to 6 should work fine right now (the upper limit is the 16kb heap pages).

#8 - 10/04/2021 08:13 PM - peterzhu2118 (Peter Zhu)

Your concerns about memory usage and performance are all valid. A solution is we could make VWA enabled by default but only default to 1 size pool (instead of the 4 size pools right now) and make the number of size pools configurable through command line flags.

This would mean very similar performance to FWA. The only difference is that we would move the string length of embedded strings from string's headers to a separate field.

Would you prefer this?

#9 - 10/04/2021 08:15 PM - ko1 (Koichi Sasada)

The performance decrease is small but the memory decrease is large.

Sorry, which results do you say? rdoc generation?

#10 - 10/04/2021 08:17 PM - peterzhu2118 (Peter Zhu)

Yes, rdoc generation. Speed is 0.95x but memory usage dropped to 0.81x (0.89x for jemalloc).

#11 - 10/04/2021 08:18 PM - ko1 (Koichi Sasada)

Ah, fool question. Sorry. It was arguments on rdoc benchmark (I misunderstood that it is for general characteristics).

#12 - 10/04/2021 11:55 PM - duerst (Martin Dürst)

This is impressive work. I haven't time to study all the details, in particular the benchmark results.

But I'm somewhat worried by:

Gems with native extensions built with VWA are not compatible with non-VWA built gems (and vice versa). When switching between VWA and non-VWA rubies, gems must be rebuilt. This is because the header file rstring.h changes depending on whether USE_RVARGC is set or not (the internal string implementation changes with VWA).

I'm not an expert on native extensions, but wouldn't this mean that we have to increase the binary compatibility version (or whatever it's called)? Wouldn't this work better if we introduced it when starting work on Ruby 3.2?

And given there are other core classes where something similar to String is planned, wouldn't that mean that we have the same problems again, and again?

If there's some mechanism in RubyGems that handles this automatically, we are probably good. If there's some mechanism in Bundler that handles this automatically, it could work. What we have to avoid at all costs is that this produces errors because Gems are not up to date or don't get recompiled when necessary. If the error can easily be detected and results in a clear error message, that might be okay, but not if it results in things such as segmentation faults.

I have put this on the agenda of the next Developers Meeting (DevelopersMeeting20211021Japan, issue #18174). But please do not stop your work as you did for #18045. That was not my intention then, and I'm sorry I missed your comment to that effect.

#13 - 10/05/2021 01:08 PM - peterzhu2118 (Peter Zhu)

Hi duerst (Martin Dürst). Thank you for adding it to the developer meeting.

I understand your concern for gem incompatibility. Since the public-facing header files often include internal implementation details, we don't expect this to be the last time we introduce an internal implementation change that breaks compatibility (e.g. arrays work very similar to strings so we expect to introduce another incompatibility when we implement VWA for arrays). We will communicate this incompatibility as much as possible through various avenues, but I think it's fair that users/developers of a developing version should expect backwards incompatible changes.

I'm not very familiar with the internal implementation of RubyGems, but from what I understand, it installs gems based on the Ruby version (major.minor.patch). So unless we bump the patch version, RubyGems will not install new gems. We've explored using prerelease flags (e.g. 3.1.0-rvargc), but RubyGems has incompatibilities and various issues with it. Also prerelease flags are not a very sustainable solution when we introduce more and more incompatible changes.

#14 - 10/05/2021 03:10 PM - Eregon (Benoit Daloze)

Regarding ABI compatibility, AFAIK CRuby doesn't care about it for development versions.
I.e., conceptually we can think of every commit potentially changing the ABI but the ABI version number (RbConfig::CONFIG['ruby_version']) is never changed for dev versions of CRuby. So it's indeed currently up to users of CRuby dev versions to basically always wipe out their compiled gems whenever rebuilding a new dev version.

FWIW TruffleRuby actually tracks the ABI of dev versions through this file which means it is possible to sensibly cache compiled gems even for dev versions.

ruby/setup-ruby has no choice for CRuby dev but to use the commit as the ABI version. This issue is made worse by Ruby switchers like RVM & chruby setting GEM_HOME (so the ABI is effectively ignored by RubyGems in those cases, and those directories need to be cleaned manually).

When GEM_HOME is not set, it would be enough to rebuild CRuby dev and remove the directory before installing (which includes both CRuby & gems), but Ruby installers don't do that yet. Bundler always includes the ABI version when setting the bundler path (bundle config --local path), but if the ABI version is incorrect like for CRuby dev it's of no use.

Anyway, I don't want to distract more about ABI for this issue. The ABI version not representing the ABI for CRuby dev versions is a long-standing issue, but orthogonal to this proposal and I believe it does not need to be solved for this feature.

#15 - 10/05/2021 06:56 PM - peterzhu2118 (Peter Zhu)
- Description updated

#16 - 10/11/2021 05:59 PM - Eregon (Benoit Daloze)
I filed a separate issue regarding the ABI version reported by CRuby: https://bugs.ruby-lang.org/issues/18249

#17 - 10/21/2021 06:02 AM - matz (Yukihiro Matsumoto)
I agree with merging it, but it's too close to the 3.1 release. So how about merging it, turned off by default, then turning it on after the release?

Matz.

#18 - 10/22/2021 08:03 PM - peterzhu2118 (Peter Zhu)
Thank you matz (Yukihiro Matsumoto). We will merge with it turned off by default for 3.1.

#19 - 10/25/2021 05:26 PM - peterzhu2118 (Peter Zhu)
- Status changed from Open to Closed

Applied in changeset git|46b66eb9e8e6de2d5750591e532310e8f8599d90.

[Feature #18239] Add struct for embedded strings