Feature description

Since merging the initial implementation in #17570, we've been working on improving the performance of the allocator (which was one of the major drawbacks in the initial implementation). We've chosen to return to using a freelist-based allocator and added support for variable slot size pages in what we call "size pools". We're still keeping the USE_RVARGC compile-time flag that maintains current behaviour when it is not explicitly enabled.

Summary

- We now use pages with different slot sizes in pools to speed up allocation. Objects will be allocated in the smallest slot size that fits the requested size. This brings us back to the freelist-based allocation algorithm and significantly increases allocation performance.
- The heap growth algorithm has been changed to adapt to the growth pattern of the size pool. Heaps that rapidly allocate and deallocate are considered "growth heaps" and are treated differently than heaps that are stable in size.

Size pools

In this patch, we introduce a new structure into the GC called "size pools". Each size pool contains an eden and tomb heap to hold the pages. The size pool determines the slot size of the pages in it. We've chosen powers of 2 multiples of RVALUE size as our slot sizes. In other words, since the RVALUE size is 40 bytes, size pool 0 will have pages with slot size 40B, 80B for size pool 1, 160B for size pool 2, 320B for size pool 3, etc. The reason we chose powers of 2 multiples of RVALUE is that powers of 2 are easy to work with (e.g. we can use bit shifts) and using multiples of RVALUE means that we do not waste space for the single RVALUE (40 byte) allocations. When VWA is not enabled, there is only one size pool.

Heap growth algorithm

This patch changes heap growth when USE_RVARGC is turned on. Heap growth is now calculated after sweeping (rather than after marking). This is because we don't know the characteristics of a size pool after marking (we only know the number of marked slots vs. total number of pages). By keeping track of the number of free slots and swept slots of each size pool during sweeping, we know the exact number of live vs. free slots in each size pool. This allows us to grow the size pool according to its growth characteristics. For example, classes are allocated in the third size pool (160B slot size). For most workloads, classes are allocated at boot and very rarely allocated afterwards. Through the data collected during sweeping, we can determine that this size pool is no longer growing and thus allow it to be very full.

Lazy sweeping

At every sweeping step, we attempt to sweep a little bit of every size pool. If the size pool we're allocating into didn't yield a page during sweeping and that size pool is not allowed to create new pages, then we must finish sweeping by sweeping all remaining pages in all other size pools. This may cause some lazy sweeping steps to take longer than others, we can see the result of this in the worse p99 response time in railsbench.

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
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Written by Roland McGrath and Ulrich Drepper.

jemalloc version:

$ apt list --installed | grep jemalloc

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

libjemalloc-dev/focal,now 5.2.1-1ubuntu1 amd64 [installed]
libjemalloc2/focal,now 5.2.1-1ubuntu1 amd64 [installed,automatic]

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

Master was benchmarked on commit 84fea8ee39249ff9e7a03c407e5d16ad93074f3e. 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

- We don't expect to see a significant performance improvement with this patch. In fact, our main goal of this patch is for Variable
  Width Allocation to have performance and memory usage comparable to master.
- railsbench:
  - VWA uses about 1.02x more memory than master.
  - When using jemalloc, VWA is about 1.027x faster than master.
  - We see no (within margin of error) speedup when using glibc.
  - In both glibc and jemalloc we see worse p99 response times described above in "Lazy sweeping" section. However, the
    p100 response times of VWA is comparable to master.
- rdoc generation:
  - VWA uses 1.13x less memory than master.
- liquid benchmarks:
  - We see no significant performance changes here.
- optcarrot benchmark:
  - We see no significant performance changes here.

railsbench

For railsbench, we ran the railsbench benchmark. For both the performance and memory benchmarks, 50 runs were conducted for each
combination (branch + glibc, master + glibc, branch + jemalloc, master + jemalloc).

glibc

For glibc, the RPS between the three are all within the margin of error. However, the p99 of the branch with VWA is significantly
worse than master (3.2ms vs 1.8ms). This is due to the longer lazy sweeping steps discussed above. However, this isn't a problem
for p100 response times.

<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>731.69</td>
<td>724.19</td>
<td>727.02</td>
</tr>
<tr>
<td>p50 (ms)</td>
<td>1.33</td>
<td>1.38</td>
<td>1.37</td>
</tr>
<tr>
<td>p66 (ms)</td>
<td>1.37</td>
<td>1.41</td>
<td>1.40</td>
</tr>
<tr>
<td>p75 (ms)</td>
<td>1.38</td>
<td>1.42</td>
<td>1.41</td>
</tr>
<tr>
<td>p80 (ms)</td>
<td>1.39</td>
<td>1.43</td>
<td>1.42</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>p90 (ms)</td>
<td>1.41</td>
<td>1.45</td>
<td>1.45</td>
</tr>
<tr>
<td>p95 (ms)</td>
<td>1.44</td>
<td>1.48</td>
<td>1.47</td>
</tr>
<tr>
<td>p98 (ms)</td>
<td>1.50</td>
<td>1.78</td>
<td>1.76</td>
</tr>
<tr>
<td>p99 (ms)</td>
<td>3.23</td>
<td>3.84</td>
<td>1.83</td>
</tr>
<tr>
<td>p100 (ms)</td>
<td>15.74</td>
<td>16.48</td>
<td>16.38</td>
</tr>
</tbody>
</table>

For memory usage, we see a slight increase in memory usage when using Variable Width Allocation compared to master.

125795979-dc006bb5-e129-47f0-8db7-23c2037ba2f2.png
Average max memory usage for VWA: 147.08 MB

Average max memory usage for master: 143.49 MB

VWA uses 1.025x more memory.

jemalloc

For jemalloc, we see that the branch is 1.027x faster than master in RPS (standard deviation is about 3 rps). We once again see the worse p99 response times with VWA enabled.

<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>756.06</td>
<td>741.26</td>
<td>736.31</td>
</tr>
<tr>
<td>p50 (ms)</td>
<td>1.30</td>
<td>1.34</td>
<td>1.34</td>
</tr>
<tr>
<td>p66 (ms)</td>
<td>1.32</td>
<td>1.36</td>
<td>1.37</td>
</tr>
<tr>
<td>p75 (ms)</td>
<td>1.33</td>
<td>1.37</td>
<td>1.39</td>
</tr>
<tr>
<td>p75 (ms)</td>
<td>1.34</td>
<td>1.38</td>
<td>1.39</td>
</tr>
<tr>
<td>p90 (ms)</td>
<td>1.36</td>
<td>1.40</td>
<td>1.41</td>
</tr>
<tr>
<td>p95 (ms)</td>
<td>1.38</td>
<td>1.42</td>
<td>1.44</td>
</tr>
<tr>
<td>p98 (ms)</td>
<td>1.42</td>
<td>1.71</td>
<td>1.70</td>
</tr>
<tr>
<td>p99 (ms)</td>
<td>2.74</td>
<td>1.78</td>
<td>1.78</td>
</tr>
<tr>
<td>p100 (ms)</td>
<td>13.42</td>
<td>16.80</td>
<td>16.45</td>
</tr>
</tbody>
</table>

Once again, we see a slight increase in memory usage in VWA compared to master.

125796015-ff2bd763-0646-44df-8b0c-46161bb425c6.png
Average max memory usage for VWA: 145.47 MB

Average max memory usage for master: 142.65 MB

VWA uses 1.020x more memory.

rdoc generation

For rdoc generation, we ran the following script for the branch and master to collect memory usage.

```bash
for x in $(seq 50)
do
  sudo rm -rf .ext/rdoc; mstat -o mstat/branch_$x.tsv -freq 59 -- ruby --disable-gems "./libexec/rdoc" --root "." --encoding=UTF-8 --all --ri --op ".ext/rdoc" --page-dir "./doc" --no-force-update "."
done
```

For rdoc generation, we see a decrease in memory usage when using Variable Width Allocation.

glibc

125796394-b6ae6041-2eb4-4e7d-9e14-e5b37e69e218.png
Average max memory usage for VWA: 328.83 MB

Average max memory usage for master: 373.17 MB
VWA uses 1.13x less memory.

jemalloc

125796437-ef9dc34b-ea66-4fa5-a0c2-051f8dc281b3.png
Average max memory usage for VWA: 308.08 MB
Average max memory usage for master: 347.07 MB
VWA uses 1.13x less memory.

Liquid benchmarks

For the liquid benchmarks, we ran the liquid benchmark averaged over 5 runs each. We don't see any significant performance improvements or regressions here.

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse (i/s)</td>
<td>39.60</td>
<td>39.56</td>
<td>39.45</td>
</tr>
<tr>
<td>Render (i/s)</td>
<td>127.69</td>
<td>126.91</td>
<td>127.06</td>
</tr>
<tr>
<td>Parse &amp; Render (i/s)</td>
<td>28.42</td>
<td>28.40</td>
<td>28.30</td>
</tr>
</tbody>
</table>

optcarrot

For optcarrot, we ran the optcarrot benchmark averaged over 5 runs each. Once again, we don't see any significant performance improvements or regressions here.

<table>
<thead>
<tr>
<th></th>
<th>Branch (VWA on)</th>
<th>Branch (VWA off)</th>
<th>Master</th>
</tr>
</thead>
<tbody>
<tr>
<td>fps</td>
<td>44.54</td>
<td>45.02</td>
<td>44.15</td>
</tr>
</tbody>
</table>

Future plans

- Improve within size pool compaction support. The compaction only currently compacts the first size pool (i.e. size pool with 40B slots).
- Improve ractor support. The ractor cache currently only caches the first size pool (size pool with 40B slots). Any >40B allocations require locking the VM.
- Increase coverage of Variable Width Allocation. Our next candidates are arrays and strings.
- Investigate ways to support resizing of objects within Variable Width Allocation.
  - One idea is to support cross size pool compaction. So when an object requests to be resized it will be moved to the most optimal size pool.
- Use powers of 2 slot sizes in heap pages. We currently use powers of 2 multiples of RVALUE size, but it would be easier and more efficient to use powers of 2 slot sizes (i.e. 32B, 64B, 128B, etc.). However, this would mean that 24B will be wasted for objects that are 40B in size (occupy a single RVALUE) since they will be allocated into 64B slots. We believe that once more objects can take advantage of Variable Width Allocation, we will be able to shrink the size of RVALUE to 32B. As such, we only plan on investigating this once most, if not all, objects can be allocated through Variable Width Allocation.

Related issues:
Related to Ruby master - Feature #17816: Move C heap allocations for RVALUE o... Closed

Associated revisions
Revision 48ff7a9f - 08/23/2021 01:15 PM - peterzhu2118 (Peter Zhu)
[Feature #18045] Remove T_PAYLOAD
This commit removes T_PAYLOAD since the new VWA implementation no longer requires T_PAYLOAD types.
Co-authored-by: Aaron Patterson tenderlove@ruby-lang.org

Revision b2e2cf2d - 08/23/2021 01:15 PM - peterzhu2118 (Peter Zhu)
[Feature #18045] Implement size classes for GC

This commits implements size classes in the GC for the Variable Width Allocation feature. Unless USE_RVARGC compile flag is set, only a single size class is created, maintaining current behaviour. See the redmine ticket for more details.

Co-authored-by: Aaron Patterson tenderlove@ruby-lang.org

Revision edd369e - 08/23/2021 02:54 PM - peterzhu2118 (Peter Zhu)
Revert "[Feature #18045] Implement size classes for GC"
This reverts commits 48ff7a9f3e47bfb9e4d067a12ba9b936251caa0 and b2e2c2dedd104acad8610721db5e4d341f13fe5 because it is causing crashes in SPARC solaris and i386 debian.

Revision c08d4067 - 08/25/2021 01:28 PM - peterzhu2118 (Peter Zhu)
[Feature #18045] Remove T_PAYLOAD
This commit removes T_PAYLOAD since the new VWA implementation no longer requires T_PAYLOAD types.

Co-authored-by: Aaron Patterson tenderlove@ruby-lang.org

Revision 62bc4a94 - 08/25/2021 01:28 PM - peterzhu2118 (Peter Zhu)
[Feature #18045] Implement size classes for GC
This commits implements size classes in the GC for the Variable Width Allocation feature. Unless USE_RVARGC compile flag is set, only a single size class is created, maintaining current behaviour. See the redmine ticket for more details.

Co-authored-by: Aaron Patterson tenderlove@ruby-lang.org

History

#1 - 08/04/2021 09:16 PM - peterzhu2118 (Peter Zhu)
Hello! We're still looking for feedback on this. If there are no concerns, we'll merge it next week.

#2 - 08/05/2021 12:55 AM - duerst (Martin Dürst)
I think this should really be discussed seriously at least once at a committer's meeting. I have added it to #18039.
Using powers of 2 slot sizes is the "trusted and true" way of handling this, but may lead to some memory overhead. What about e.g. using powers of two, but keep a separate slot size for 40 bytes, so that RVALUEs don't incur overhead? Also, it may be worth at this time to do a pass through the various object types (TArray, TString,...) and figuring out cases where what's currently is RVALUE can be dealt with in 32 bytes, or will benefit from being extended to 64 bytes, or can otherwise be improved or risk some additional overhead.

#3 - 08/05/2021 12:56 AM - duerst (Martin Dürst)
- Related to Feature #17816: Move C heap allocations for RVALUE object data into GC heap added

#4 - 08/05/2021 02:52 PM - peterzhu2118 (Peter Zhu)
Hi duerst (Martin Dürst), thank you for the feedback. Thank you for adding it to the committer meeting, we will wait for their feedback before proceeding further.

Using powers of 2 slot sizes is the "trusted and true" way of handling this, but may lead to some memory overhead.

One of our end goals is to use powers of 2 slot sizes by either (1) reducing RVALUE size to 32 bytes, or (2) enough types use VWA such that 40 byte allocations become less frequent. We don't anticipate (1) or (2) to happen in the near future since we still need to tackle other issues first (e.g. resizing).

What about e.g. using powers of two, but keep a separate slot size for 40 bytes, so that RVALUEs don't incur overhead?

Since VWA is turned off by default (must be compiled with USE_RVARGC=1 to enable VWA), we compromise by using powers of 2 multiples of RVALUE. This is "close enough" to powers of 2 slot sizes while we don't need to handle special cases for 40 byte RVALUEs. Since would like to build this incrementally (i.e. many small patches rather than a single large patch), we would like to avoid special cases that lead to dead code when VWA is turned off.
Also, it may be worth at this time to do a pass through the various object types (TArray, TString,...) and figuring out cases where what's currently is RVALUE can be dealt with in 32 bytes, or will benefit from being extended to 64 bytes, or can otherwise be improved or risk some additional overhead.

We are optimistic about Arrays and Strings being the next types to use VWA since past results in #9362 have shown to increase their performance when RVALUE size is extended to 64 bytes. We'll continue to look for opportunities and ways to move to powers of 2 slot sizes.

#5 - 08/23/2021 01:16 PM - peterzhu2118 (Peter Zhu)
- Status changed from Open to Closed

Applied in changeset gill48ff7a9f3e47bfff3e4d067a12ba9b936261caa0.

[Feature #18045] Remove T_PAYLOAD

This commit removes T_PAYLOAD since the new VWA implementation no longer requires T_PAYLOAD types.

Co-authored-by: Aaron Patterson tenderlove@ruby-lang.org