Move C heap allocations into GC heap for RVALUE object data

01/22/2021 04:10 PM - eightbitraptor (Matthew Valentine-House)

Status: Closed
Priority: Normal
Assignee:
Target version:

Description

Feature

Move C heap allocations into GC heap.

Pull Request:

4107

Feature description

RVALUE is fixed width (40 bytes), some of which is used for accounting information and the remainder being used for storage of the object data (e.g. string contents). If the data required is larger than the available space inside the RVALUE, memory is allocated outside of the GC heap using malloc and a pointer to the malloc'd region is stored inside the RVALUE instead.

We intend to remove this extra memory allocation in favour of storing the extra data in a contiguous region of slots in the heap that are adjacent to the original RVALUE.

The proposed memory layout changes in this feature branch can be seen in the following diagrams.

Original layout (master branch - with extra data being stored in malloc regions):

RXNvZkK.png

New layout (feature branch - with extra data being stored adjacent to it's owner in the GC heap):

oRVgdkJ.png

Our hypothesis is that this feature will improve performance across Ruby by

- Improving locality, allowing better cache performance.
- Increased memory efficiency as fragmented data outside of GC can now be controlled and compacted by the GC.
- Reducing pointer indirection - as the extra object data (eg the Class ivar table) is in a predictable place, we can find it by using the object address plus some offset rather than hopping over pointers.
- Eventually allowing us to remove the transient heap - The transient heap exists to reduce the number of these "extra data" malloc allocations. For new objects, the extra data is stored in a separate, pre-allocated heap, and then transferred to the C heap using memcpyp once the data owner becomes an old object (survives 3 GC runs). This type of performance optimisation becomes redundant if we can store all object data in the eden heap.

Current status

This branch is the minimum viable implementation required to test our hypothesis. We introduce an API to allocate regions that are multiple slots wide in the heap and we implement this API for objects of type T_CLASS so that the rb_classext_t struct is stored in the heap alongside its owning RVALUE.

There are many things left to implement and some large things to fix. All of which will be addressed in future work, which we're working on at Shopify.

- We have consciously traded allocation performance for simplicity in this initial approach as read performance is where we think there will be a significant gain.
- GC Compaction is incompatible with this change. The current two finger algorithm is designed to support fixed width memory regions. When an object occupies multiple contiguous slots this algorithm no longer works.
- Incremental marking is disabled when this feature is enabled. This is because we are unable to predict the allocation pattern in each mark step now that allocations can be multiple slots wide.
- We cannot currently resize regions. For example, when mutating strings, we currently realloc one region to a new larger region,
we will support resizing when all object data is in the eden heap.
- This API is currently only implemented for T_CLASS objects, we need to implement it for all other Ruby types to gain the most benefit.

We believe this feature is useful enough that we'd like to merge this PR. We would like feedback from the Core team and the community.

Merging this feature will also allow us to continue working without maintaining a long lived fork of Ruby.

There will be no negative effects of merging this PR as all functionality is disabled by default.

**How to enable this feature**

This feature must be enabled at compile time with the flag USE_RVARGC. E.g.

```sh
cppflags='-DUSE_RVARGC=1' ./configure
make clean && make && make install
```

**Performance**

We ran the following snippet to generate a heap dump and then used a separate script to visualise it.

```ruby
require "objspace"
ObjectSpace.dump_all(output: File.open("heap.json", "w"), full: true)
```

In the following output each 2x2 pixel region represents a single slot in a heap page. Heap pages are organised vertically and from low to high memory address.

![sXYqJQ.png](attachment:sXYqJQ.png)

- Blue pixels represents T_CLASS
- Yellow pixels represents T_PAYLOAD slots
- Grey pixels are other live objects

So we can see that every T_CLASS is now followed by 3 T_PAYLOAD objects - this is because sizeof(rb_classext_t) == 104 and the space available in 3 T_PAYLOAD slots is 112 (we require 8 bytes for bookkeeping).

We prepared a simple micro-benchmark in order to measure the speed of ivar access and method calls, as references to the method table and the ivar table are stored inside rb_classext_struct, which is always allocated on the malloc heap.

```ruby
require 'benchmark/ips'

class Foobar
  def initialize
    @foo = 3
  end

  def foo
    @foo
    self
  end

  def bar
    @foo + 1
    self
  end

  def baz
    @foo + 2
    self
  end

  @f = Foobar.new

Benchmark.ips do |x|
  x.report("foo")
  x.report("bar")
  x.report("baz")
end
```

How to enable this feature

This feature must be enabled at compile time with the flag USE_RVARGC. E.g.

```sh
cppflags='-DUSE_RVARGC=1' ./configure
make clean && make && make install
```

**Performance**

We ran the following snippet to generate a heap dump and then used a separate script to visualise it.

```ruby
require "objspace"
ObjectSpace.dump_all(output: File.open("heap.json", "w"), full: true)
```

In the following output each 2x2 pixel region represents a single slot in a heap page. Heap pages are organised vertically and from low to high memory address.

![sXYqJQ.png](attachment:sXYqJQ.png)

- Blue pixels represents T_CLASS
- Yellow pixels represents T_PAYLOAD slots
- Grey pixels are other live objects

So we can see that every T_CLASS is now followed by 3 T_PAYLOAD objects - this is because sizeof(rb_classext_t) == 104 and the space available in 3 T_PAYLOAD slots is 112 (we require 8 bytes for bookkeeping).

We prepared a simple micro-benchmark in order to measure the speed of ivar access and method calls, as references to the method table and the ivar table are stored inside rb_classext_struct, which is always allocated on the malloc heap.

```ruby
require 'benchmark/ips'

class Foobar
  def initialize
    @foo = 3
  end

  def foo
    @foo
    self
  end

  def bar
    @foo + 1
    self
  end

  def baz
    @foo + 2
    self
  end

  @f = Foobar.new

Benchmark.ips do |x|
  x.report("foo")
  x.report("bar")
  x.report("baz")
end
```
x.report("method & ivar lookup") do |times|
  i = 0
  while i < times
    @f.foo.bar.baz
    i += 1
  end
end
end

We ran this 5 times on our branch and took the scores, and then again on master (at 32b7dcfb56a417c1d1c354102351fc1825d653bf - the base of our feature branch) and took the scores.

This graph shows the mean of our results combined with the standard error range for each set of samples.

84GJ7CS.png

Micro-benchmark: Million iterations per second, higher is better

master:
  16.372
  16.399
  16.058
  15.98
  16.426

feature:
  16.901
  16.909
  16.695
  16.757
  16.818

This shows a ~3% improvement in our feature-branch over master on average, with the worst case (upper bound of standard error on master vs lower bound standard error on our feature branch) showing a ~2% improvement.

We also ran the optcarrot benchmark on each branch and the results were as follows:

Q8EQRoE.png

Optcarrot: Frames per second, higher is better

master:
  43.27430926
  42.65007047
  43.32562887
  42.64047665
  43.40328494

feature:
  41.59069356
  40.51252649
  40.55757381
  41.76160937
  42.95645122

This shows a that our branch is ~4% slower than master (with the branch upper vs master lower standard error difference being ~2%).

Raw benchmark numbers and the script used for generating these charts is in this gist.

Despite the Optcarrot performance drop we feel that these results are promising for the future of this work - taking into consideration our significantly worse allocation performance (improving allocation performance is a high priority part of our roadmap).

History
#1 - 01/22/2021 04:12 PM - eightbitraptor (Matthew Valentine-House)
  - Description updated
Thank you for the great work.

It is interesting idea.
At first glance, “GC Compaction” and “Incremental marking” seem major drawbacks, I'll see the patch more.

Thank you for the great work.

It is interesting idea.
At first glance, “GC Compaction” and “Incremental marking” seem major drawbacks, I'll see the patch more.

Thank you for taking the time to look.

We're definitely aware that the impact to Compaction and incremental marking are major drawbacks. We will address those as a future part of this work. We pushed this knowing about the drawbacks so that we could make other developers aware of our intention and get some feedback on the idea.

This has been superseded by the work in this ticket.

Status changed from Open to Closed