Ruby - Feature #18776

Object Shapes

05/11/2022 09:02 PM - jemmai (Jemma Issroff)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

Object Shapes implementation

Aaron Patterson, Eileen Uchitelle and I have been working on an implementation of Object Shapes for Ruby. We are filing a ticket to share what we've been doing, as well as get feedback on the project in its current state.

We hope to eventually submit the finalized project upstream after verifying efficacy.

What are Object Shapes?

Object shapes are a technique for representing properties of an object. Other language implementations, including TruffleRuby and V8, use this technique. Chris Seaton, the creator of TruffleRuby, discussed object shapes in his RubyKaigi 2021 Keynote and Maxime Chevalier-Boisvert discussed the implications for YJIT in the latter part of her talk at RubyKaigi 2021. The original idea of object shapes originates from the Self language, which is considered a direct descendant of Smalltalk.

Each shape represents a specific set of attributes (instance variables and other properties) and their values. In our implementation, all objects have a shape. The shapes are nodes in a tree data structure. Every edge in the tree represents an attribute transition.

More specifically, setting an instance variable on an instance of an object creates an outgoing edge from the instance's current shape. This is a transition from one shape to another. The edge represents the instance variable that is set.

For example:

```
class Foo
  def initialize
    # Currently this instance is the root shape (ID 0)
    @a = 1 # Transitions to a new shape via edge @a (ID 1)
    @b = 2 # Transitions to a new shape via edge @b (ID 2)
  end
end

foo = Foo.new
```

When Foo is intialized, its shape is the root shape with ID 0. The root shape represents an empty object with no instance variables. Each time an instance variable is set on foo, the shape of the instance changes. It first transitions with @a to a shape with ID 1, and then transitions with @b to a shape with ID 2. If @a is then set to a different value, its shape will remain the shape with ID 2, since this shape already includes the instance variable @a.

167918360-0a6c91aa-2587-48cb-8ff2-7f3a9583288e.svg
There is one global shape tree and objects which undergo the same shape transitions in the same order will end up with the same final shape.

For instance, if we have a class Bar defined as follows, the first transition on Bar.new through the instance variable @a will be the same as Foo.new's first transition:

```
class Foo
  def initialize
    # Currently this instance is the root shape (ID 0)
    @a = 1 # Transitions to a new shape via edge @a (ID 1)
    @b = 2 # Transitions to a new shape via edge @b (ID 2)
  end
end
```

```
class Bar
  def initialize
    # Currently this instance is the root shape (ID 0)
    @a = 1 # Transitions to shape defined earlier via edge @a (ID 1)
    @c = 1 # Transitions to a new shape via edge @c (ID 3)
    @b = 1 # Transitions to a new shape via edge @b (ID 4)
  end
end

foo = Foo.new # blue in the diagram
bar = Bar.new # red in the diagram
```

In the diagram below, purple represents shared transitions, blue represents transitions for only foo, and red represents transitions for only bar.

167918899-f1a6f344-ae5e-4dc0-b17a-fb156d1d550f.svg

Cache structure

For instance variable writers, the current shape ID, the shape ID that ivar write would transition to and instance variable index are all stored in the inline cache. The shape ID is the key to the cache.

For instance variable readers, the shape ID and instance variable index are stored in the inline cache. Again, the shape ID is the cache key.

```
class Foo
  def initialize
    @a = 1 # IC shape_id: 0, next shape: 1, iv index 0
    @b = 1 # IC shape_id: 1, next shape: 2, iv index 1
  end

  def a
    @a # IC shape_id: 2, iv index 0
  end
end
```

Rationale

We think that an object shape implementation will simplify cache checks, increase inline cache hits, decrease runtime checks, and enable other potential future optimizations. These are all explained below.

Simplify caching

The current cache implementation depends on the class of the receiver. Since the address of the class can be reused, the current cache implementation also depends on an incrementing serial number set on the class (the class serial). The shape implementation has no such dependency. It only requires checking the shape ID to determine if the cache is valid.

Cache hits

Objects that set properties in the same order can share shapes. For example:

```
class Hoge
  def initialize
    # Currently this instance is the root shape (ID 0)
    @a = 1 # Transitions to the next shape via edge named @a
    @b = 2 # Transitions to next shape via edge named @b
  end
end

class Fuga < Hoge; end

hoge = Hoge.new
fuga = Fuga.new
```

In the above example, the instances hoge and fuga will share the same shape ID. This means inline caches in initialize will hit in both cases. This contrasts with the current implementation that uses the class as the cache key. In other words, with object shapes the above code will hit inline caches where the current implementation will miss.

If performance testing reveals that cache hits are *not* substantially improved, then we can use shapes to reclaim memory from RBasic. We can accomplish this by encoding the class within the shape tree. This will have an equivalent cache hit rate to the current implementation. Once the class is encoded within the shape tree, we can remove the class pointer from RBasic and either reclaim that memory or free it for another use.

Decreases runtime checking

We can encode attributes that aren't instance variables into an object's shape. Currently, we also include frozen within the shape. This means we can limit frozen checks to only cache misses.

For example, the following code:

```
class Toto
  def set_a
    @a = 1 # cache: shape: 0, next shape: 1, IV idx: 0
  end
end

toto = Toto.new # shape 0
toto.set_a      # shape 1

toto = Toto.new # shape 0
toto.freeze     # shape 2
toto.set_a      # Cache miss, Exception!
```

167920001-c4e6326b-3a3c-483b-a797-9e02317647d7.svg
Without shapes, all instance variable sets require checking the frozen status of the object. With shapes, we only need to check the frozen status on cache misses.

We can also eliminate embedded and extended checks with the introduction of object shapes. Any particular shape represents an object that is *either* extended or embedded. JITs can possibly take advantage of this fact by generating specialized machine code based on the shapes.

Class instance variables can be stored in an array

Currently, T_CLASS and T_MODULE instances cannot use the same IV index table optimization that T_OBJECT instances use. We think that the object shapes technique can be leveraged by class instances to use arrays for class instance variable storage and may possibly lead to a reduction in memory usage (though we have yet to test this).

Implementation Details

Here is a link to our code so far.

As mentioned earlier, shape objects form a tree data structure. In order to look up the shape quickly, we also store the shapes in a weak array that is stored on the VM. The ID of the shape is the index in to the array, and the ID is stored on the object.

For T_OBJECT objects, we store the shape ID in the object's header. On 64 bit systems, the upper 32 bits are used by Ractors. We want object shapes to be enabled on 32 bit systems and 64 bit systems so that limits us to the bottom 32 bits of the Object header. The top 20 bits for T_OBJECT objects was unused, so we used the top 16 bits for the shape id. We chose the top 16 bits because this allows us to use uint16_t and masking the header bits is easier.

This implementation detail limits us to ~65k shapes. We measured the number of shapes used on a simple Discourse application (~3.5k), RailsBench (~4k), and Shopify's monolith's test suite (~40k). Accordingly, we decided to implement garbage collection on object shapes so we can recycle shape IDs which are no longer in use. We are currently working on shape GC.

Even though it's unlikely, it's still possible for an application to run out of shapes. Once all shape IDs are in use, any objects that need new shape IDs will never hit on inline caches.

Evaluation

We have so far tested this branch with Discourse, RailsBench and Shopify's monolith. We plan to test this branch more broadly against several open source Ruby and Rails applications.

Before we consider this branch to be ready for formal review, we want the runtime performance and memory usage of these benchmarks to be equivalent or better than it is currently. In our opinion, even with equivalent runtime performance and memory usage, the future benefits of this approach make the change seem worthwhile.

Feedback

If you have any feedback, comments, or questions, please let us know and we'll do our best to address it. Thank you!

Associated revisions

Revision 9ddfd2ca004d1952be79cf1b84c52c79a55978f4 - 09/26/2022 04:21 PM - jemmai (Jemma Issroff)

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the
"frozenness" of objects. Object instances have a "shape" and the shape
represents some attributes of the object (currently which instance variables are
set and the "frozenness"). Shapes form a tree data structure, and when a new
instance variable is set on an object, that object "transitions" to a new shape
in the shape tree. Each shape has an ID that is used for caching. The shape
structure is independent of class, so objects of different types can have the
same shape.

For example:

```
class Foo
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set
instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more
efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See
RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org
Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Revision 9ddfd2ca004d1952be79cf1b84c52c79a55978f4 - 09/26/2022 04:21 PM - jemmai (Jemma Issroff)

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the
"frozenness" of objects. Object instances have a "shape" and the shape
represents some attributes of the object (currently which instance variables are
set and the "frozenness"). Shapes form a tree data structure, and when a new
instance variable is set on an object, that object "transitions" to a new shape
in the shape tree. Each shape has an ID that is used for caching. The shape
structure is independent of class, so objects of different types can have the
same shape.

For example:

```
class Foo
  def initialize
```

```
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org
Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Revision 9ddfd2ca - 09/26/2022 04:21 PM - jemmai (Jemma Issroff)

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the "frozenness" of objects. Object instances have a "shape" and the shape represents some attributes of the object (currently which instance variables are set and the "frozenness"). Shapes form a tree data structure, and when a new instance variable is set on an object, that object "transitions" to a new shape in the shape tree. Each shape has an ID that is used for caching. The shape structure is independent of class, so objects of different types can have the same shape.

For example:

```
class Foo
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org
Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Revision d594a5a8bd0756f65c078fcf5ce0098250cba141 - 09/28/2022 03:26 PM - jemmai (Jemma Issroff)

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the "frozenness" of objects. Object instances have a "shape" and the shape represents some attributes of the object (currently which instance variables are set and the "frozenness"). Shapes form a tree data structure, and when a new instance variable is set on an object, that object "transitions" to a new shape in the shape tree. Each shape has an ID that is used for caching. The shape structure is independent of class, so objects of different types can have the same shape.

For example:

```
class Foo
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org
Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Revision d594a5a8bd0756f65c078fcf5ce0098250cba141 - 09/28/2022 03:26 PM - jemmai (Jemma Issroff)

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the "frozenness" of objects. Object instances have a "shape" and the shape represents some attributes of the object (currently which instance variables are set and the "frozenness"). Shapes form a tree data structure, and when a new instance variable is set on an object, that object "transitions" to a new shape in the shape tree. Each shape has an ID that is used for caching. The shape structure is independent of class, so objects of different types can have the same shape.

For example:

```
class Foo
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set
instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more
efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See
RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org
Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Revision d594a5a8 - 09/28/2022 03:26 PM - jemmai (Jemma Issroff)

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the
"frozenness" of objects. Object instances have a "shape" and the shape
represents some attributes of the object (currently which instance variables are
set and the "frozenness"). Shapes form a tree data structure, and when a new
instance variable is set on an object, that object "transitions" to a new shape
in the shape tree. Each shape has an ID that is used for caching. The shape
structure is independent of class, so objects of different types can have the
same shape.

For example:

```
class Foo
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set
instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more
efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See
RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org

Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Revision 99825a539f990dff7e0d7cae082ab195a82ffaa5 - 07/17/2024 02:25 PM - alanwu (Alan Wu)

[DOC] Note that rb_obj_freeze_inline() can raise NoMemoryError

And move it back to a public header because Doxygen might not be
scanning the .c files.

[Feature #18776]

Revision 99825a539f990dff7e0d7cae082ab195a82ffaa5 - 07/17/2024 02:25 PM - alanwu (Alan Wu)

[DOC] Note that rb_obj_freeze_inline() can raise NoMemoryError

And move it back to a public header because Doxygen might not be
scanning the .c files.

[Feature #18776]

Revision 99825a53 - 07/17/2024 02:25 PM - alanwu (Alan Wu)

[DOC] Note that rb_obj_freeze_inline() can raise NoMemoryError

And move it back to a public header because Doxygen might not be
scanning the .c files.

[Feature #18776]

History

#1 - 05/11/2022 09:04 PM - tenderlovemaking (Aaron Patterson)

- Description updated

#2 - 05/11/2022 11:54 PM - tenderlovemaking (Aaron Patterson)

- Description updated

#3 - 05/12/2022 01:24 AM - ko1 (Koichi Sasada)

Great patch.
I'm looking forward to seeing evaluation results.

Questions:

- how to use parent id?
- how to find next id with additional ivar?

#4 - 05/12/2022 12:41 PM - byroot (Jean Boussier)

> We want object shapes to be enabled on 32 bit systems and 64 bit systems so that limits us to the bottom 32 bits of the Object header.

Might be a silly question, but how popular are 32bits systems these days? Would it be acceptable to make objects a bit bigger on 32 bits systems so that both 32bits and 64bits Ruby have a 32bit shape IDs?

#5 - 05/12/2022 03:56 PM - jemmai (Jemma Issroff)

Thanks for the feedback, Koichi.

ko1 (Koichi Sasada) wrote in #note-3:

> - how to use parent id?

The rb_shape type is as follows:

```
struct rb_shape {
    VALUE flags;
    struct rb_shape * parent;
    struct rb_id_table * edges;
    struct rb_id_table * iv_table;
    ID edge_name;
};
```

parent is a pointer on the shape itself, and we use it primarily for GC

> - how to find next id with additional ivar?

When we add a new ivar, if there is no transition we have to find a new ID. Right now we're doing a linear scan of available IDs. Once we're confident in shape ID GC, we'll switch the algorithm to something more efficient like using a bitmap

#6 - 05/12/2022 03:59 PM - tenderlovemaking (Aaron Patterson)

byroot (Jean Boussier) wrote in #note-4:

> > We want object shapes to be enabled on 32 bit systems and 64 bit systems so that limits us to the bottom 32 bits of the Object header.
>
> Might be a silly question, but how popular are 32bits systems these days? Would it be acceptable to make objects a bit bigger on 32 bits systems so that both 32bits and 64bits Ruby have a 32bit shape IDs?

I'm not sure how popular 32 bit systems are, especially where high performance is a requirement. But I do think the broader question of "how much work should we do to support 32 bit systems?" is something we should think about. For now I think we can make shapes work well on 32 bit machines so I'm not too worried about it at this point (though it definitely would be easier if we had more than 16 bits ▯▯)

#7 - 05/12/2022 06:19 PM - byroot (Jean Boussier)

> it definitely would be easier if we had more than 16 bits

Yeah, my worry is that while the Shopify monolith is probably among the bigger codebases, ~40k out of ~65k is really not that much leeway.

#8 - 05/12/2022 06:46 PM - tenderlovemaking (Aaron Patterson)

byroot (Jean Boussier) wrote in #note-7:

> > it definitely would be easier if we had more than 16 bits
>
> Yeah, my worry is that while the Shopify monolith is probably among the bigger codebases, ~40k out of ~65k is really not that much leeway.

When we measured on Shopify core it was before we had started implementing shape GC. There were ~40k shapes, but that was the total number of shapes ever seen. Hopefully with shape GC we'll see a lower number of live shapes.

#9 - 05/12/2022 08:05 PM - masterleep (Bill Lipa)

If you call memoized methods in a different order, would that cause instances of the same class to have multiple shapes?

#10 - 05/12/2022 08:20 PM - byroot (Jean Boussier)

> If you call memoized methods in a different order, would that cause instances of the same class to have multiple shapes?

Yes.

#11 - 05/13/2022 01:06 AM - ko1 (Koichi Sasada)

jemmai (Jemma Issroff) wrote in #note-5:

> When we add a new ivar, if there is no transition we have to find a new ID. Right now we're doing a linear scan of available IDs. Once we're confident in shape ID GC, we'll switch the algorithm to something more efficient like using a bitmap

Ah, my question was how to find an existing transition.

#12 - 05/13/2022 01:11 AM - ko1 (Koichi Sasada)

struct rb_id_table * edges;

and I understand edges manages the next transitions. Thanks.

#13 - 08/17/2022 05:14 PM - jemmai (Jemma Issroff)

- File object-shapes.patch added

Object Shapes Update

We are writing with an update on the Object Shapes implementation, and to ask what needs to be done before we can merge our work. I have continued work on this alongisde Aaron and Eileen.

Code changes

These are our proposed code changes to implement Object Shapes in CRuby.

This patch adds an object shape implementation. Each object has a shape, which represents attributes of the object, such as which slots ivars are stored in and whether objects are frozen or not. The inline caches are updated to use shape IDs as the key, rather than the class of the object. This means we don't have to read the class from the object to check IC validity. It also allows more cache hits in some cases, and will allow JITs to optimize instance variable reading and writing.

The patch currently limits the number of available shape IDs to 65,536 (using 16 bits). We created a new IMEMO type that represents the shape, so shapes can be garbage collected. Collected shape IDs can be reused later.

CPU performance:

We measured performance with microbenchmarks, RailsBench, and YJIT bench. Here are the performance metrics we gathered.

These are all microbenchmarks which measure ivar performance:

```
$ make benchmark ITEM=vm_ivar
compare-ruby: ruby 3.2.0dev (2022-08-16T15:58:56Z master ac890ec062) [arm64-darwin21]
built-ruby: ruby 3.2.0dev (2022-08-16T20:12:55Z object-shapes-prot.. 872fa488c3) [arm64-darwin21]
# Iteration per second (i/s)

|                           |compare-ruby|built-ruby|
|:--------------------------|-----------:|---------:|
|vm_ivar                    |     98.231M|  102.161M|
|                           |           -|     1.04x|
|vm_ivar_embedded_obj_init  |     33.351M|   33.331M|
|                           |       1.00x|         -|
|vm_ivar_extended_obj_init  |     25.055M|   26.265M|
|                           |           -|     1.05x|
|vm_ivar_generic_get        |     18.374M|   17.215M|
|                           |       1.07x|         -|
|vm_ivar_generic_set        |     12.361M|   14.537M|
|                           |           -|     1.18x|
|vm_ivar_of_class           |      8.378M|    8.928M|
|                           |           -|     1.07x|
|vm_ivar_of_class_set       |      9.485M|   10.264M|
|                           |           -|     1.08x|
|vm_ivar_set                |     89.411M|   91.632M|
|                           |           -|     1.02x|
|vm_ivar_init_subclass      |      6.104M|   12.928M|
|                           |           -|     2.12x|
```

To address the outliers above:

- vm_ivar_generic_set is faster because this patch adds inline caches to generic ivars, which did not exist previously
- vm_ivar_init_subclass is significantly faster because, with shapes, subclasses can hit caches (as class is no longer part of the cache key)

Object Shapes and Ruby master perform roughly the same on RailsBench.

On the following measurement, Ruby master had 1852.1 requests per second, while Object Shapes had 1842.7 requests per second.

```
$ RAILS_ENV=production bin/bench
ruby 3.2.0dev (2022-08-15T14:00:03Z master 0264424d58) [arm64-darwin21]
1852.1

$ RAILS_ENV=production bin/bench
ruby 3.2.0dev (2022-08-15T15:20:22Z object-shapes-prot.. d3dbefd6cd) [arm64-darwin21]
```

```
1842.7
```

Memory performance

Each Ruby object contains a shape ID. The shape ID corresponds to an index in an array. We can easily look up the shape object given a shape ID. Currently, we have a fixed size array which stores pointers to all active shapes (or NULL in the case that the shape is yet to be used). That array is ~64k * sizeof(uintptr_t) (about 500kb) and is currently a fixed size overhead for the Ruby process.

Running an empty Ruby script, we can see this overhead. For instance:

On Ruby master:

```
$ /usr/bin/time -l ruby -v -e' '
ruby 3.2.0dev (2022-08-15T14:00:03Z master 0264424d58) [arm64-darwin21]
28639232  maximum resident set size
```

With the shapes branch:

```
$ /usr/bin/time -l ./ruby -v -e' '
ruby 3.2.0dev (2022-08-15T15:20:22Z object-shapes-prot.. d3dbefd6cd) [arm64-darwin21]
28917760  maximum resident set size
```

This is roughly a 0.97% memory increase on an empty Ruby script. Obviously, on bigger Ruby processes, it would represent an even smaller memory increase.

YJIT Statistics

We also ran YJIT-bench and got the following results:

on Ruby master:

```
end_time="2022-08-17 09:31:36 PDT (-0700)"
yjit_opts=""
ruby_version="ruby 3.2.0dev (2022-08-16T15:58:56Z master ac890ec062) [x86_64-linux]"
git_branch="master"
git_commit="ac890ec062"

-------------  -----------  ----------  ---------  ----------  -----------  ------------
bench          interp (ms)  stddev (%)  yjit (ms)  stddev (%)  interp/yjit  yjit 1st itr
30k_ifelse     2083.0       0.1         203.6      0.0         10.23        0.80
30k_methods    5140.1       0.0         476.7      0.1         10.78        3.95
activerecord   188.1        0.1         99.5       0.2         1.89         1.23
binarytrees    804.8        0.1         409.2      1.1         1.97         1.93
cfunc_itself   232.5        2.4         43.3       1.5         5.36         5.34
chunky_png     2316.9       0.2         757.3      0.3         3.06         2.86
erubi          412.1        0.4         281.3      1.0         1.46         1.47
erubi_rails    31.1         2.2         17.4       2.7         1.78         0.33
fannkuchredux  11414.6      0.2         2773.5     1.3         4.12         1.00
fib            591.8        1.1         41.7       4.5         14.20        13.93
getivar        234.2        3.1         23.5       0.1         9.95         1.00
hexapdf        4755.7       1.0         2517.3     3.0         1.89         1.51
keyword_args   520.7        0.6         54.6       0.2         9.55         9.24
lee            2274.1       0.2         1133.3     0.2         2.01         1.98
liquid-render  296.7        0.3         139.3      2.8         2.13         1.46
mail           212.9        0.1         127.9      0.1         1.66         0.72
nbody          225.4        0.2         78.3       0.2         2.88         2.70
optcarrot      14592.1      0.7         4072.8     0.3         3.58         3.43
psych-load     3947.8       0.0         2075.5     0.1         1.90         1.88
railsbench     2826.0       0.6         1774.4     1.9         1.59         1.26
respond_to     424.3        0.2         154.5      3.1         2.75         2.76
rubykon        22545.1      0.4         6993.5     1.3         3.22         3.24
setivar        185.9        5.6         97.0       0.0         1.92         1.00
str_concat     123.1        0.9         28.6       2.0         4.31         3.35
-------------  -----------  ----------  ---------  ----------  -----------  ------------
Legend:
- interp/yjit: ratio of interp/yjit time. Higher is better. Above 1 represents a speedup.
- 1st itr: ratio of interp/yjit time for the first benchmarking iteration.
```

with the shapes branch:

```
end_time="2022-08-16 13:56:32 PDT (-0700)"
yjit_opts=""
ruby_version="ruby 3.2.0dev (2022-08-15T18:35:34Z object-shapes-prot.. 51a23756c3) [x86_64-linux]"
git_branch="object-shapes-prototyping"
```

```
git_commit="51a23756c3"
```

```
-------------  ------------  ----------  ---------  ----------  -----------  ------------
bench          interp (ms)   stddev (%)  yjit (ms)  stddev (%)  interp/yjit  yjit 1st itr
30k_ifelse     2135.2        0.0         340.1      0.1         6.28         0.95
30k_methods    5180.7        0.0         906.2      0.1         5.72         3.56
activerecord   189.2         0.1         174.5      0.1         1.08         0.83
binarytrees    783.2         1.0         438.7      2.5         1.79         1.82
cfunc_itself   225.2         1.6         44.0       0.6         5.11         5.01
chunky_png     2394.9        0.2         1657.0     0.2         1.45         1.44
erubi          418.1         0.5         284.3      1.1         1.47         1.45
erubi_rails    31.6          1.5         26.2       2.1         1.21         0.34
fannkuchredux  12208.5       0.1         2821.6     0.4         4.33         0.99
fib            565.7         0.3         41.3       0.1         13.69        13.59
getivar        247.6         0.1         244.9      2.0         1.01         1.02
hexapdf        4961.0        1.6         4926.1     0.9         1.01         0.94
keyword_args   499.7         0.8         57.0       0.4         8.77         8.65
lee            2360.0        0.6         2138.6     0.6         1.10         1.11
liquid-render  294.7         0.7         274.9      1.4         1.07         0.91
mail           216.6         0.1         157.7      0.7         1.37         0.70
nbody          232.7         0.2         237.2      0.5         0.98         0.99
optcarrot      15095.8       0.7         18309.2    0.5         0.82         0.83
psych-load     4174.5        0.1         3707.9     0.1         1.13         1.13
railsbench     2923.7        0.8         2548.4     1.4         1.15         0.98
respond_to     409.2         0.3         162.6      1.7         2.52         2.52
rubykon        22554.1       0.7         20160.6    0.9         1.12         1.10
setivar        249.6         0.1         169.5      0.1         1.47         0.99
str_concat     137.8         0.8         29.0       2.4         4.75         3.50
-------------  ------------  ----------  ---------  ----------  -----------  ------------
Legend:
- interp/yjit: ratio of interp/yjit time. Higher is better. Above 1 represents a speedup.
- 1st itr: ratio of interp/yjit time for the first benchmarking iteration.
```

We are seeing some variations in YJIT benchmark numbers, and are working on addressing them.

32 bit architectures

We're storing the shape ID for T_OBJECT types in the top 32 bits of the flags field (sharing space with the ractor ID). Consequently 32 bit machines do not benefit from this patch. This patch makes 32 bit machines always miss on inline caches.

Instance variables with ID == 0

This is minor, but we also do not support instance variables whose ID is 0 because the outgoing edge tables are id_tables which do not support 0 as a key. There is one test for this feature, and we have marked it as pending in this patch.

Merging

We think this feature is ready to merge. Please give us feedback, and let us know if it is possible to merge now. If it's not possible, please let us know what needs to be improved so that we can merge.

Future work

We plan to work next on speeding up the class instance variables. We will implement caching for this, and see the full benefits of object shapes in this case.

#14 - 08/17/2022 06:05 PM - maximecb (Maxime Chevalier-Boisvert)

> These are our proposed code changes to implement Object Shapes in CRuby.

I think it would be a good idea to open a draft pull request so that it's easier to look at the diff and comment on it.

> We also ran YJIT-bench and got the following results:

Can you guys investigate a bit why there are large slowdowns with YJIT?

I believe you said you had written code to make use of object shapes in YJIT. I would have expected the performance difference to be small. Since it's so big,
I suspect that maybe there are a large number of side-exits happening, or something like that.

We could pair over it tomorrow if that's helpful. It's probably not difficult to fix.

#15 - 08/17/2022 07:03 PM - jemmai (Jemma Issroff)

> I think it would be a good idea to open a draft pull request so that it's easier to look at the diff and comment on it.

Good idea, [here is a draft PR]() and updated it above as well

> Can you guys investigate a bit why there are large slowdowns with YJIT?

We realized we accidentally ran these in debug mode. We will get new numbers and repost them, sorry about that!

#16 - 08/17/2022 07:51 PM - jemmai (Jemma Issroff)

After running the YJIT benchmarks in release mode, we found that setting instance variables is, indeed, slower on our branch than the master branch.

YJIT is not exiting. YJIT is calling out to rb_vm_setinstancevariable to set instance variables. We disassembled this function on both master and our branch and it looks like the compiler is emitting unfortunate machine code on our branch. It looks like we are spilling data to the stack where we weren't in master.

185229672-d0d7e1e5-9897-4673-a9e4-8460165cefce.png
On the left is the machine code in our branch, on the right is the machine code for master. It looks like the machine code on our branch is spilling data to the stack and we think this is why there's a speed difference. We can work on changing the C code to get more efficient machine code.

#17 - 08/17/2022 08:32 PM - maximecb (Maxime Chevalier-Boisvert)

It's unfortunate that there are spills there and there might be ways to reduce that by reorganizing the code a bit, but I would expect the performance impact of the spills to be relatively small, because in practice in most kinds of software, there are many times more ivar reads than ivar writes, and those spills are a drop in the bucket.

Also, if you look at the getivar benchmark before and after, you can see that the speedup went from 9.95 to 1.01. That suggests that we're probably side-exiting or not running the getinstancevariable code in YJIT for some reason. IMO getinstancevariable should be the first place to look.

#18 - 08/18/2022 04:50 AM - naruse (Yui NARUSE)

About the general principle, since Ruby is used on many production environment by many companies, non-optional feature needs to be production ready.

It means it shouldn't have a downside including CPU performance, Memory consumption, and so on. Since YJIT is already concerned as a production feature, it's also a downside if it cause slow down of YJIT. Also note that increasing code complexity is also a downside.

Though a critical downside needs to be fixed, a small downside can be accepted if it has a larger upside. For example code complexity, it is accepted while it introduces a enough performance improvement compared to the complexity.

As far as I understand, the Object Shape is introduce to improve performance and it's complex. Therefore it needs to show some performance improvement compared to the code complexity before we merge it to ruby-master.

#19 - 08/18/2022 03:40 PM - Dan0042 (Daniel DeLorme)

Thank you for this important work. In particular I think shapes will be very useful in the future to improve the performance of keyword arguments.

The following comments/nitpicks may be irrelevant (or I may have misunderstood the code completely), but I had to get them off my brain:

(1) It would be nice to convert the magic numbers into constants. So maybe we could have #define SHAPE_BITS 16 and #define SHAPE_MASK ((1<<SHAPE_BITS)-1) and then other constants that derive from that, and maybe it would be possible to compile a version with 17-bit shapes.

(2) It looks to me like rb_shape_get_iv_index is really critical to performance, but the linked-list structure has bad performance characteristics for locality of memory access. If an object has 6 ivars you need to read 7 non-contiguous structs for *each* ivar access. It should be much faster if each shape struct had the full list of IDs in the shape. With a SIMD instruction it would be super-fast to find the index for a given ID, but even without SIMD the memory locality would be much better.

(3) The frozen flag is represented as a different root shape, but this results in many extra shapes that are not used by any object:

```
def initialize(a,b,c,d)
  @a,@b,@c,@d = a,b,c,d
  freeze
end
```

```
0(root) -> 2(@a) -> 3(@a,@b) -> 4(@a,@b,@c) -> 5(@a,@b,@c,@d)
```

```
1(frozen) -> 6(@a) -> 7(@a,@b) -> 8(@a,@b,@c) -> 9(@a,@b,@c,@d)
```

If the frozen flag was represented by a leaf node, this would use fewer shapes. (It would also mirror the order of operations and the fact that after freezing it's not possible add more ivars.)

```
0(root) -> 1(@a) -> 2(@a,@b) -> 3(@a,@b,@c) -> 4(@a,@b,@c,@d)
                                                |
                                                5(@a,@b,@c,@d,frozen)
```

#20 - 08/18/2022 05:38 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote in #note-19:

> In particular I think shapes will be very useful in the future to improve the performance of keyword arguments.

Can you explain how shapes could improve performance of keyword arguments? Nobody else has mentioned the possibility yet, and I'm not sure how they are related.

> (3) The frozen flag is represented as a different root shape, but this results in many extra shapes that are not used by any object:
>
> ```
> def initialize(a,b,c,d)
>   @a,@b,@c,@d = a,b,c,d
>   freeze
> end
> ```
>
> ```
> 0(root) -> 2(@a) -> 3(@a,@b) -> 4(@a,@b,@c) -> 5(@a,@b,@c,@d)
> ```
>
> ```
> 1(frozen) -> 6(@a) -> 7(@a,@b) -> 8(@a,@b,@c) -> 9(@a,@b,@c,@d)
> ```

If an object is frozen, you cannot add instance variables to it. I don't see how you could generate shapes 6-9 in your example. I would guess the implementation already works similarly to the leaf node approach you suggested.

#21 - 08/18/2022 06:06 PM - tenderlovemaking (Aaron Patterson)

Dan0042 (Daniel DeLorme) wrote in #note-19:

> Thank you for this important work. In particular I think shapes will be very useful in the future to improve the performance of keyword arguments.

I don't think this will have any impact on keyword arguments.

> (2) It looks to me like rb_shape_get_iv_index is really critical to performance, but the linked-list structure has bad performance characteristics for locality of memory access. If an object has 6 ivars you need to read 7 non-contiguous structs for *each* ivar access. It should be much faster if each shape struct had the full list of IDs in the shape. With a SIMD instruction it would be super-fast to find the index for a given ID, but even without SIMD the memory locality would be much better.

rb_shape_get_iv_index is rarely called because the index is stored in the inline cache. We only need to call this when the cache misses. It does make cache misses expensive in terms of CPU, but memory is reduced.

> (3) The frozen flag is represented as a different root shape, but this results in many extra shapes that are not used by any object:
>
> ```
> def initialize(a,b,c,d)
>   @a,@b,@c,@d = a,b,c,d
>   freeze
> end
> ```
>
> ```
> 0(root) -> 2(@a) -> 3(@a,@b) -> 4(@a,@b,@c) -> 5(@a,@b,@c,@d)
> ```
>
> ```
> 1(frozen) -> 6(@a) -> 7(@a,@b) -> 8(@a,@b,@c) -> 9(@a,@b,@c,@d)
> ```

This case can't happen because you can't add ivars after an object has been frozen.

We have a "frozen root shape" just as an optimization for objects that go from the root shape and are immediately frozen. For example when using the frozing_string_literals directive. Objects that are not T_OBJECT, T_CLASS, or T_MODULE store their shape id in the gen iv table. We didn't want to make a geniv table for every object that goes from root -> frozen (again think of the number of frozen string literals in an application), so we pre-allocate that shape, then assign it "at birth" (using the frozen bit to indicate that we're using the "frozen root shape" singleton).

> If the frozen flag was represented by a leaf node, this would use fewer shapes. (It would also mirror the order of operations and the fact that after freezing it's not possible add more ivars.)
>
> ```
> 0(root) -> 1(@a) -> 2(@a,@b) -> 3(@a,@b,@c) -> 4(@a,@b,@c,@d)
>                                                 |
>                                                 5(@a,@b,@c,@d,frozen)
> ```

It's currently implemented this way. :)

naruse (Yui NARUSE) wrote in #note-18:

> About the general principle, since Ruby is used on many production environment by many companies, non-optional feature needs to be production ready.
>
> It means it shouldn't have a downside including CPU performance, Memory consumption, and so on. Since YJIT is already concerned as a production feature, it's also a downside if it cause slow down of YJIT. Also note that increasing code complexity is also a downside.
>
> Though a critical downside needs to be fixed, a small downside can be accepted if it has a larger upside. For example code complexity, it is accepted while it introduces a enough performance improvement compared to the complexity.
>
> As far as I understand, the Object Shape is introduce to improve performance and it's complex. Therefore it needs to show some performance improvement compared to the code complexity before we merge it to ruby-master.

We're working to fix the YJIT cases. I think we can see higher speeds in YJIT using Object Shapes than the current caching mechanisms as the machine code YJIT generates can be much more simple.

For the non-JIT case, we're seeing good improvements in some benchmarks like this:

```
|vm_ivar_init_subclass       |       6.104M|    12.928M|
|                            |            -|      2.12x|
```

This is a case where shapes can hit inline caches but the current mechanism cannot, like this code:

```
class A
  def initialize
    @a = 1
    @b = 1
  end
end

class B < A; end

loop do
  A.new
  B.new
end
```

Our current cache cannot hit because the classes change, but shapes *can* hit because the shapes are the same.

As for complexity, I think the strategy is easier to understand than our current caching mechanism and it actually simplifies cache checks. We only need to compare shape id, not class serial + frozen status. Code complexity is hard to measure though, and I admit this patch is pretty big. ▯▯

#22 - 08/18/2022 06:22 PM - chrisseaton (Chris Seaton)

Reference keyword arguments - what we're doing is using the same idea of shapes, but applying them to the keyword arguments hash to be able to extract keyword arguments without any control-flow. We can implement keyword arguments like this with just a single machine-word comparison overhead over positional arguments.

https://www.youtube.com/watch?v=RVqY1FRUm_8

#23 - 08/18/2022 06:49 PM - Dan0042 (Daniel DeLorme)

tenderlovemaking (Aaron Patterson) wrote in #note-21:

> It's currently implemented this way. :)

Ok, thank you for the explanation. I got completely confused by the role of the frozen root shape. Sorry for the noise.

jeremyevans0 (Jeremy Evans) wrote in #note-20:

> Can you explain how shapes could improve performance of keyword arguments? Nobody else has mentioned the possibility yet, and I'm not sure how they are related.

Well, I believe shapes can be considered a general-purpose mechanism to store "named tuples" or "hash of symbols" kind of data structures.

Keyword arguments like foo(a:1, b:2) are already pretty efficient because they use a strategy similar to shapes (VM_CALL_KWARG) where keys {a,b} are stored once per-callsite and values [1,2] are passed on the stack (IIRC).

But as soon as you have a splat you need to allocate a hash on the heap. (VM_CALL_KW_SPLAT)

```
h = {c:3, d:4}
foo(a:1, b:2, **h)
```

With shapes you could start with {a,b} and then add the hash keys to get shape {a,b,c,d} and pass all values [1,2,3,4] plus the shape id on the stack. No need to allocate a hash and associated st_table/ar_table. I think.

#24 - 08/18/2022 06:50 PM - chrisseaton (Chris Seaton)

> With shapes you could start with {a,b} and then add the hash keys to get shape {a,b,c,d} and pass all values [1,2,3,4] plus the shape id on the stack.

Yes that's the idea in the video I linked.

#25 - 09/15/2022 07:23 PM - jemmai (Jemma Issroff)

Summary

The implementation has been updated to solve some performance problems and simplify both source code and generated code.

The performance of this branch against multiple benchmarks, including microbenchmarks, RailsBench and YJIT Bench show that the performance of Object Shapes is equivalent to, if not better than, the existing ivar implementation.

We have also improved the memory overhead and made it half of what it was previously.

Overall, code complexity has been decreased, memory overhead is small and performance is better or on par with master. We think this is a good state for this feature to be merged. We would like to get feedback from Ruby committers on this PR or in this issue for that reason.

Details

Since our previous update, we have made the following changes:

- The shape ID is now 32 bits on 64 bit machines when debugging is disabled. This gives us significantly more shapes in most cases. When debugging is enabled, Ractor check mode is also enabled. Ractor check mode stores the Ractor ID in the flags bits, and shapes needs to use that space too. Given this constraint, when debug mode is enabled Shapes only consumes 16 bits leaving the other 16 bits for Ractor IDs.
- The shape ID is now stored in the upper 32 bits of the flags field. This ensures a consistent location for finding the shape id. All objects allocated from the heap will have their shape id stored in the top 32 bits. This simplifies looking up the shape id for a given object, which reduces code complexity. It also simplifies the machine code emitted from the JIT.
- On platforms that support mmap, we now use mmap to allocate the shape list. Since mmap lazily maps to physical pages, this allows us to lazily allocate space for the shape list.

CPU performance

These are our updated perfomance results:

Microbenchmarks

We ran microbenchmarks comparing master to object shapes and got the following results:

```
$  make benchmark ITEM=vm_ivar
...
compare-ruby: ruby 3.2.0dev (2022-09-13T06:44:29Z master 316b44df09) [arm64-darwin21]
built-ruby: ruby 3.2.0dev (2022-09-13T11:25:59Z object-shapes-prot.. ef42354c33) [arm64-darwin21]
# Iteration per second (i/s)

|                           |compare-ruby|built-ruby|
|:--------------------------|-----------:|---------:|
|vm_ivar                    |    100.020M|  108.705M|
|                           |           -|     1.09x|
|vm_ivar_embedded_obj_init  |     33.584M|   33.415M|
|                           |       1.01x|         -|
|vm_ivar_extended_obj_init  |     26.073M|   26.635M|
|                           |           -|     1.02x|
|vm_ivar_generic_get        |     16.599M|   18.103M|
|                           |           -|     1.09x|
|vm_ivar_generic_set        |     12.505M|   18.616M|
|                           |           -|     1.49x|
|vm_ivar_get                |       8.533|     8.566|
|                            |          -|     1.00x|
|vm_ivar_get_uninitialized   |    81.117M|   79.294M|
|                            |      1.02x|         -|
|vm_ivar_lazy_set            |      1.921|     1.949|
|                            |          -|     1.01x|
|vm_ivar_of_class            |     8.359M|    9.094M|
|                            |          -|     1.09x|
|vm_ivar_of_class_set        |    10.678M|   10.331M|
|                            |      1.03x|         -|
|vm_ivar_set                 |    90.398M|   92.034M|
|                            |          -|     1.02x|
|vm_ivar_set_on_instance     |     14.269|    14.307|
|                            |          -|     1.00x|
|vm_ivar_init_subclass       |     6.048M|   13.029M|
|                            |          -|     2.15x|
```

Class instance variables have never benefited from inline caching, so always take the non-cached slow path. Object shapes made the slow path slower, so the vm_ivar_of_class_set benchmark slowdown is expected.

As follow up to object shapes, @jhawthorn (John Hawthorn) is planning to re-implement instance variables on classes to use arrays instead of st_tables. With his change, class instance variables will be able to realize the benefits of object shapes by taking the cached, fast path.

Railsbench

We ran Railsbench 10 times on master and shapes, and got the following results:

Master:
BMeML1Q.png

```
ruby 3.2.0dev (2022-09-13T06:44:29Z master 316b44df09) [arm64-darwin21]
Request per second: 1898.3 [#/s] (mean)
Request per second: 1890.8 [#/s] (mean)
Request per second: 1894.7 [#/s] (mean)
Request per second: 1885.0 [#/s] (mean)
Request per second: 1868.2 [#/s] (mean)
Request per second: 1884.2 [#/s] (mean)
Request per second: 1860.6 [#/s] (mean)
Request per second: 1902.1 [#/s] (mean)
Request per second: 1927.1 [#/s] (mean)
Request per second: 1894.8 [#/s] (mean)
```

This averages to 1890.58 requests per second

Shapes:

```
ruby 3.2.0dev (2022-09-13T11:25:59Z object-shapes-prot.. ef42354c33) [arm64-darwin21]
Request per second: 1901.9 [#/s] (mean)
Request per second: 1900.1 [#/s] (mean)
Request per second: 1903.1 [#/s] (mean)
Request per second: 1902.2 [#/s] (mean)
Request per second: 1905.2 [#/s] (mean)
Request per second: 1903.0 [#/s] (mean)
Request per second: 1910.7 [#/s] (mean)
Request per second: 1916.3 [#/s] (mean)
Request per second: 1905.6 [#/s] (mean)
Request per second: 1894.4 [#/s] (mean)
```

This averages to 1904.25 requests per second

YJIT Bench:

We ran YJIT Bench on master and shapes, excluding benchmarks which do not measure instance variables, and got the following results:

Master:

```
ruby_version="ruby 3.2.0dev (2022-09-13T06:44:29Z master 316b44df09) [x86_64-linux]"
git_branch="master"
git_commit="316b44df09"
```

```
-------------  -----------  ----------  ---------  ----------  -----------  ------------
bench          interp (ms)  stddev (%)  yjit (ms)  stddev (%)  interp/yjit  yjit 1st itr
activerecord   115.5        0.3         74.7       2.1         1.55         1.31
chunky_png     748.0        0.1         500.9      0.1         1.49         1.46
erubi          260.4        0.6         202.8      1.0         1.28         1.25
erubi_rails    18.4         2.0         13.2       3.5         1.39         0.50
getivar        91.9         1.4         23.1       0.2         3.98         0.98
hexapdf        2183.0       0.9         1479.4     3.0         1.48         1.31
liquid-render  144.3        0.4         87.6       1.5         1.65         1.32
mail           125.6        0.2         104.5      0.3         1.20         0.81
optcarrot      5013.9       0.7         2227.7     0.5         2.25         2.20
psych-load     1804.1       0.1         1333.0     0.0         1.35         1.35
railsbench     1933.3       1.1         1442.5     1.6         1.34         1.20
rubykon        9838.4       0.4         4915.1     0.2         2.00         2.11
setivar        64.9         1.4         27.9       3.1         2.32         1.01
```

Shapes:

```
ruby_version="ruby 3.2.0dev (2022-09-13T11:25:59Z object-shapes-prot.. ef42354c33) [x86_64-linux]"
git_branch="object-shapes-prototyping"
git_commit="ef42354c33"
```

```
-------------  -----------  ----------  ---------  ----------  -----------  ------------

bench          interp (ms)  stddev (%)  yjit (ms)  stddev (%)  interp/yjit  yjit 1st itr
activerecord   118.6        0.1         76.4       0.2         1.55         1.27
chunky_png     760.5        0.2         488.8      0.3         1.56         1.52
erubi          252.4        0.6         199.9      1.0         1.26         1.25
erubi_rails    18.5         2.5         13.7       3.3         1.35         0.53
getivar        89.8         1.2         23.3       0.0         3.85         1.00
hexapdf        2364.9       1.0         1649.2     2.8         1.43         1.30
liquid-render  147.3        0.6         90.3       1.7         1.63         1.30
mail           128.7        0.3         106.0      0.2         1.21         0.82
optcarrot      5170.7       0.8         2681.7     0.3         1.93         1.88
psych-load     1786.4       0.1         1480.2     0.0         1.21         1.20
railsbench     1988.9       0.8         1482.4     1.7         1.34         1.23
rubykon        9729.6       1.2         4841.3     1.7         2.01         2.17
setivar        61.6         0.3         32.2       0.1         1.91         1.00
```

Here are comparison numbers between the two measurements. (> 1 means shapes is faster, < 1 means master is faster):

```
bench   interp (shapes / master)    yjit (shapes / master)
activerecord  1.03                       1.02
chunky_png  1.02                       0.98
erubi   0.97                        0.99
erubi_rails  1.01                        1.04
getivar   0.98                       1.01
hexapdf   1.08                       1.11
liquid-render  1.02                       1.03
mail   1.02                        1.01
optcarrot  1.03                       1.20
psych-load  0.99                       1.11
railsbench  1.03                       1.03
rubykon   0.99                       0.98
setivar   0.95                       1.15
```

Memory consumption

Due to the mmap change, our memory consumption has decreased since our last update. Measuring execution of an empty script over 10 runs, we saw an average consumption of 29,107,814 bytes on master, and 29,273,293 bytes with object shapes. This means on an empty script, shapes has a 0.5% memory increase. Obviously, this difference would represent a significantly lower percentage memory increase on larger, production-scale applications.

Code complexity

We have reduced overall code complexity by:

- Removing the iv_index table on the class and replacing it with a shape tree which can be used independent of object types
- Reducing the checks in the set instance variable and get instance variable cached code paths by removing the frozen check, class serial check and vm_ic_entry_p check in favor of a shape check.

The code below is the fast code path (cache hit case) for setting instance variables. (Assertions and debug counters removed for clarity.)

Master:

```
if (LIKELY(!RB_OBJ_FROZEN_RAW(obj))) {
    if (LIKELY(
        (vm_ic_entry_p(ic) && ic->entry->class_serial == RCLASS_SERIAL(RBASIC(obj)->klass)))) {

        if (UNLIKELY(index >= ROBJECT_NUMIV(obj))) {
            rb_init_iv_list(obj);
        }
        VALUE *ptr = ROBJECT_IVPTR(obj);
        RB_OBJ_WRITE(obj, &ptr[index], val);
        return val;
    }
}
```

Shapes:

```
shape_id_t shape_id = ROBJECT_SHAPE_ID(obj);

if (shape_id == source_shape_id) {
    if (dest_shape_id != shape_id) {
        if (UNLIKELY(index >= ROBJECT_NUMIV(obj))) {
            rb_init_iv_list(obj);
        }
        ROBJECT_SET_SHAPE_ID(obj, dest_shape_id);
    }

    VALUE *ptr = ROBJECT_IVPTR(obj);
    RB_OBJ_WRITE(obj, &ptr[index], val);
    return val;
}
```

- Reducing the number of instructions for instance variable access in YJIT. Here are the x86_64 assembly code instructions generated by YJIT which represent guard comparisons for getting instance variables:

Master:

```
# guard known class
0x55cfff14857a: movabs rcx, 0x7f3e1fceb0f8
```

```
0x55cfff148584: cmp qword ptr [rax + 8], rcx
0x55cfff148588: jne 0x55d007137491
0x55cfff14858e: mov rax, qword ptr [r13 + 0x18]
# Is the IV in range?
0x55cfff148592: cmp qword ptr [rax + 0x10], 0
0x55cfff148597: jbe 0x55d007137446
# guard embedded getivar
# Is object embedded?
0x55cfff14859d: test word ptr [rax], 0x2000
0x55cfff1485a2: je 0x55d0071374aa
```

Shapes:

```
# guard shape, embedded, and T_*
0x55a89f8c7cff: mov rcx, qword ptr [rax]
0x55a89f8c7d02: movabs r11, 0xffff00000000201f
0x55a89f8c7d0c: and rcx, r11
0x55a89f8c7d0f: movabs r11, 0x58000000002001
0x55a89f8c7d19: cmp rcx, r11
0x55a89f8c7d1c: jne 0x55a8a78b5d4e
```

The shapes code has one comparison and one memory read whereas the master code has three comparisons and four memory reads.

Conclusion

As we said at the beginning of this update, based on the metrics and rationale described above, we think object shapes is ready to merge. Please give us feedback on our PR or this issue.

Thank you!

#26 - 09/16/2022 07:55 PM - maximecb (Maxime Chevalier-Boisvert)

The performance numbers look good and I'm very happy with the improvements that you've made wrt shape ids being 32 bits. It's also nice to see that the generated code for property accesses in YJIT is shorter than before.

There seems to be a (likely minor) bug in the PR that should be fixed before we merge but besides that the PR looks to me like it's in a mergeable state, it delivers on what was promised.

#27 - 09/26/2022 06:55 PM - jemmai (Jemma Issroff)

- Status changed from Open to Closed

Applied in changeset git|9ddfd2ca004d1952be79cf1b84c52c79a55978f4.

This commit implements the Object Shapes technique in CRuby.

Object Shapes is used for accessing instance variables and representing the
"frozenness" of objects. Object instances have a "shape" and the shape
represents some attributes of the object (currently which instance variables are
set and the "frozenness"). Shapes form a tree data structure, and when a new
instance variable is set on an object, that object "transitions" to a new shape
in the shape tree. Each shape has an ID that is used for caching. The shape
structure is independent of class, so objects of different types can have the
same shape.

For example:

```
class Foo
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end

class Bar
  def initialize
    # Starts with shape id 0
    @a = 1 # transitions to shape id 1
    @b = 1 # transitions to shape id 2
  end
end
```

```
foo = Foo.new # `foo` has shape id 2
bar = Bar.new # `bar` has shape id 2
```

Both foo and bar instances have the same shape because they both set instance variables of the same name in the same order.

This technique can help to improve inline cache hits as well as generate more efficient machine code in JIT compilers.

This commit also adds some methods for debugging shapes on objects. See RubyVM::Shape for more details.

For more context on Object Shapes, see [Feature: #18776]

Co-Authored-By: Aaron Patterson tenderlove@ruby-lang.org
Co-Authored-By: Eileen M. Uchitelle eileencodes@gmail.com
Co-Authored-By: John Hawthorn john@hawthorn.email

Files

object-shapes.patch	180 KB	08/17/2022	jemmai (Jemma Issroff)