Ruby - Feature #11420

Introduce ID key table into MRI

08/06/2015 08:54 AM - ko1 (Koichi Sasada)

Status:	Closed
Priority:	Normal
Assignee:	ko1 (Koichi Sasada)
Target version:	

Description

Let's introduce ID key table to optimization.

Background

Now, most of tables are implemented by st_table.
st_table is O(1) hash data structure.

MRI uses ID keys tables for many purpose like:

- method tables (mtbl) for classes/modules
- constant table for classes/modules
- attribute (index) tables for classes/modules/...
  and so on.

Generally st_table is reasonable performance (speed/memory consumption), but
st_table has several drawbacks for ID key table.

- Need two words per entry for ordering (st_table_entry::olist). (memory)
- Need hash value (st_table_entry::hash). (memory)
- Need to call a function to compare IDs. (speed)

I think introducing ID key table data structure is reasonable.

Already, Yura Sokolov proposed sa_table #6962 for this purpose.
However, we can't introduce it because of several reasons.

(1) expose with st table (public API)
(2) sa_table assume ID as 32bit integer.

I need to explain (2) more. There were two reasons.

Biggest issue was Symbol GC. We planed to introduce Symbol GC.
At that time, we didn't decide we use sequential number for IDs created from dynamic symbols.
Actually, the first implementation of Symbol GC uses corresponding Symbol's object_id to represent ID.

Another issue was the available ID numbers.
ID has several fields (3 bits), so we can make 2^(32-3) IDs == 536,870,912 IDs.
We need at least (8+40)B per ID, so 536,870,912 IDs consumes 171,798,691,840 = 171GB.
It is huge, but we can't decide it is enough for future.

However, now the above issues are solved.

All IDs has sequential number from 1 (0 is reserved as undefined).
So that the current bit pattern of ID is [sequential number (64-3 or 32-3 bits)] [flags (3 bits)].

We can convert ID to [sequential number] and [sequential number] to ID.
So ID key table only keep [sequential number] as a key, instead of ID itself.
With this technique, we can keep 2^32 IDs.
Maybe it is enough because 2^32 IDs need at least 1,374,389,534,720 = 1.3TB.
(or not enough in a few years later?)

Anyway, using this technique, we can keep more IDs in 32bit integer.

Implementation

I want to introduce "struct rb_id_table" data structure for MRI interpreter private data structure.

id_table.h declare only common functions.
https://github.com/ko1/ruby/blob/6682a72d0f46ab3ae4c069a9e534dc0c050363f7/id_table.h

Each functions can be implemented any algorithms.

I wrote several variations.

(1) Using st (same as current implementation)
(2) Using simple array
(3) Using simple array with sequential number of ID instead of ID itself. Linear search for look up.
(4) Using simple array with sequential number of ID instead of ID itself, and sorted. Binary search for look up.
(5) Using funny falcon's Coalesced Hashing implementation [Feature #6962]

Generally, array implementation seems bad idea, but my implementation for (3), (4) uses dense array of 32bit integers, so that cache locality is nice when it is enough small table (16 entries == 64B).

Here is a small benchmark to measure memory consumption:

```
require 'objspace'

def make_class method_num
  Class.new{
    method_num.times{|i|
      define_method("m#{i}"){
      }
    }
  }
end

C = make_class(10_000)
p ObjectSpace.memsize_of(C)
__END__
```

```
 # of methods   |      10K     20K
---------------------------------
(1) st          : 496,560 992,944
(2) list(ID key): 262,296 524,440
(4) list(sorted): 196,760 393,368
(5) CHashing    : 262,296 524,440
                          (Bytes)
```

I found that most of method tables are 0 or a few entries.

method_number.png
This is a survey of method number on simple rails app (development mode).
X axis shows method number and Y axis shows the number of classes/modules which have method number == X-axis.

This picture shows that 3500 classes/modules has 0 methods.
Only a few classes/modules has over 100 methods.

So combination with array (for small number of entries) and hash (for larger number of entries) will be nice.

Now I replace type of m_tbl to this struct.
See all of implementation.
https://github.com/ruby/ruby/compare/trunk...ko1:mtbl

Conclusion

We can compete algorithms :)

Related issues:

Related to Ruby - Feature #6962: Use lighter hash structure for methods table...	**Closed**

Related to Ruby - Feature #11414: Relax ID table ordering	**Closed**	
Related to Ruby - Feature #9638: [PATCH] limit IDs to 32-bits on 64-bit systems	**Rejected**	**03/14/2014**

Associated revisions

Revision c35ff11ae516421809e0d03c278576a70fda45c4 - 08/12/2015 08:43 AM - ko1 (Koichi Sasada)

- id_table.h: introduce ID key table.
  [Feature #11420]
  This table only manage ID->VALUE table to reduce overhead of st.
  Some functions prefixed rb_id_table_* are provided.
- id_table.c: implement rb_id_table_*.
  There are several algorithms to implement it.
  Now, there are roughly 4 types:
  - st
  - array
  - hash (implemented by Yura Sokolov)
  - mix of array and hash
    The macro ID_TABLE_IMPL can choose implementation.
    You can see detailes about them at the head of id_table.c.
    At the default, I choose 34 (mix of list and hash).
    This is not final decision.
    Please report your suitable parameters or
    your data structure.
  - symbol.c: introduce rb_id_serial_t and rb_id_to_serial()
    to represent ID by serial number.
  - internal.h: use id_table for method tables.
  - class.c, gc.c, marshal.c, vm.c, vm_method.c: ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@51541 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision c35ff11a - 08/12/2015 08:43 AM - ko1 (Koichi Sasada)

- id_table.h: introduce ID key table.
  [Feature #11420]
  This table only manage ID->VALUE table to reduce overhead of st.
  Some functions prefixed rb_id_table_* are provided.
- id_table.c: implement rb_id_table_*.
  There are several algorithms to implement it.
  Now, there are roughly 4 types:
  - st
  - array
  - hash (implemented by Yura Sokolov)
  - mix of array and hash
    The macro ID_TABLE_IMPL can choose implementation.
    You can see detailes about them at the head of id_table.c.
    At the default, I choose 34 (mix of list and hash).
    This is not final decision.
    Please report your suitable parameters or
    your data structure.
  - symbol.c: introduce rb_id_serial_t and rb_id_to_serial()
    to represent ID by serial number.
  - internal.h: use id_table for method tables.
  - class.c, gc.c, marshal.c, vm.c, vm_method.c: ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@51541 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

History

#1 - 08/06/2015 08:56 AM - ko1 (Koichi Sasada)

- Related to Feature #6962: Use lighter hash structure for methods table, instance variable positions, constants added

#2 - 08/06/2015 08:56 AM - ko1 (Koichi Sasada)

- Related to Feature #11414: Relax ID table ordering added

#3 - 08/06/2015 12:08 PM - matz (Yukihiro Matsumoto)

Go ahead and experiment the idea.

Matz.

#4 - 08/06/2015 01:12 PM - ngoto (Naohisa Goto)

- Related to Feature #9638: [PATCH] limit IDs to 32-bits on 64-bit systems added

#5 - 08/06/2015 01:42 PM - ngoto (Naohisa Goto)

Indeed, I'm using machines with 2TB or 3TB main memory, and
theoretically the upper limit of 1,374,389,534,720 = 1.3TB
can be reached today.
(though this may be very rare case in practice)

I think to prepare a compile-time option to extend ID bits is enough for now.

> We need at least (8+40)B per ID, so 536,870,912 IDs consumes 171,798,691,840 = 171GB.

Is "(8+40)B" a typo of (8*40)B ?

#6 - 08/06/2015 02:02 PM - funny_falcon (Yura Sokolov)

Koichi Sasada , i've made another one "hash" for your experiments
https://github.com/ko1/ruby/pull/1

#7 - 08/06/2015 06:18 PM - ko1 (Koichi Sasada)

On 2015/08/06 23:02, funny.falcon@gmail.com wrote:

> Koichi Sasada , i've made another one "hash" for your experiments

Thank you!
Which implementation do you like?

--
// SASADA Koichi at atdot dot net

#8 - 08/06/2015 06:54 PM - funny_falcon (Yura Sokolov)

> Which implementation do you like?

The one which will be faster.

Quadratic probing is simpler, so if it is not slower (or with in couple of percents)
than coalesced chaining in usual application (big rails application :)), then it is
certainly preferred.

If per-class method cache were merged/implemented, then sorted array will be enough
(probably).

--
Sokolov Yura aka funny_falcon

#9 - 08/11/2015 01:02 PM - ko1 (Koichi Sasada)

I added `mix' data structure.
https://github.com/ko1/ruby/blob/f965b9bb3fc42cebb0dc30461a44bcf8fb550452/id_table.c

I measured another script.

```
$classes = []
M = 40
10_000.times{|i|
  defs = (1..M).map{|e|
    "def m#{e}; end"
  }.join("\n")
  eval <<-EOS
  class C#{i}
    $classes << self
    #{defs}
  end
  EOS
}
```

```
require 'objspace'
p $classes.reduce(0){|r, klass| r + ObjectSpace.memsize_of(klass)}
```

```
                                      method# (M)
IMPL#                 0               5              20              40              80
1:st          7,040,000       7,040,000       16,480,000      26,080,000      45,280,000
11:list       5,280,000       6,560,000       10,400,000      15,520,000      25,760,000
12:list       5,280,000       6,240,000       9,120,000       12,960,000      20,640,000
21:hash       5,360,000       6,640,000       10,480,000      15,600,000      25,840,000
22:hash       5,360,000       6,640,000       10,480,000      15,600,000      25,840,000
31:mix        5,360,000       6,320,000       9,200,000       25,840,000      25,840,000
                                                                              (Bytes)
```

Basically, we don't need to care about size for practical cases (small amount of methods) :p
I will commit them soon.

#10 - 08/11/2015 01:11 PM - ko1 (Koichi Sasada)

- File measurement2.png added

measurement2.png

#11 - 08/11/2015 04:20 PM - funny_falcon (Yura Sokolov)

Koichi Sasada, did you measure performance of mix approach compared to hash?
Perhaps, threshold should be set to lower value than 32.

#12 - 08/11/2015 08:16 PM - ko1 (Koichi Sasada)

- File microbenchmark.pdf added

> Koichi Sasada, did you measure performance of mix approach compared to hash?

I made micro benchmark set.

```
require 'objspace'
require 'benchmark'

$results = []
insert_perf = []
hit_perf1 = []
hit_perf2 = []
miss_perf1 = []
miss_perf2 = []
srand(0)

0.step(to: 100, by: 10){|m|
  classes = []

  defs = (1..m).map{|e|
    "def m#{e}; end"
  }.sort_by{rand}.join("\n")

  insert_perf << Benchmark.measure{
    10_000.times{|i|
      classes << Class.new{
        eval(defs)
        def method_missing mid
        end
      }
    }
  }.real

  objs = classes.map{|klass| klass.new}
  ms = (1..m).map{|i| "m#{i}".to_sym}
  miss_ms = (1..m).map{|i| "miss_m#{i}".to_sym}

  hit_perf1 << Benchmark.measure{
    10.times{
      objs.each{|obj|
        n = rand(m)
```

```
        obj.send(ms[n]) if m > 0
      }
    }
  }.real

  hit_perf2 << Benchmark.measure{
    objs.each{|obj|
      10.times{
        n = rand(m)
        obj.send(ms[n]) if m > 0
      }
    }
  }.real

  miss_perf1 << Benchmark.measure{
    10.times{
      objs.each{|obj|
        n = rand(m)
        obj.send(miss_ms[n]) if m > 0
      }
    }
  }.real

  miss_perf2 << Benchmark.measure{
    objs.each{|obj|
      10.times{
        n = rand(m)
        obj.send(miss_ms[n]) if m > 0
      }
    }
  }.real
  $results << classes.reduce(0){|r, klass| r + ObjectSpace.memsize_of(klass)}
}

puts "insert\t#{insert_perf.join("\t")}"
puts "hit_perf1\t#{hit_perf1.join("\t")}"
puts "hit_perf2\t#{hit_perf2.join("\t")}"
puts "miss_perf1\t#{miss_perf1.join("\t")}"
puts "miss_perf2\t#{miss_perf2.join("\t")}"
puts "mem\t#{$results.join("\t")}"
```

raw data is https://docs.google.com/spreadsheets/d/1NEbb6p663rc6-6xLZyM2cZQqm1kr96FyCQsYGe6KfS0/edit?usp=sharing

Charts are attached in one pdf file.

- st is slow.
- surprisingly, other methods are not so different (at least under or equal 100 methods).
  - I expected that insertion is slow on list, but not so slow.
  - I expected that lookup is fast on small list, but not so different.
- maybe we can make faster implementation (sophisticate).

Any verifications / reproducible benchmarks are welcome.
Discourse benchmark?

#13 - 08/11/2015 08:20 PM - ko1 (Koichi Sasada)

Ah, I have important notice.

Now, I'm checking method table with Ruby's method definitions and Ruby's method invocations. They are heavy operations, so that they can hide overhead of table operations (but st was slow :p).

So if we apply this table to other purpose, we can see other effect.

#14 - 08/12/2015 08:44 AM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset r51541.

- id_table.h: introduce ID key table.
  [Feature #11420]
  This table only manage ID->VALUE table to reduce overhead of st.
  Some functions prefixed rb_id_table_* are provided.
- id_table.c: implement rb_id_table_*.

There are several algorithms to implement it.
Now, there are roughly 4 types:

- st
- array
- hash (implemented by Yura Sokolov)
- mix of array and hash
  The macro ID_TABLE_IMPL can choose implementation.
  You can see detailes about them at the head of id_table.c.
  At the default, I choose 34 (mix of list and hash).
  This is not final decision.
  Please report your suitable parameters or
  your data structure.
- symbol.c: introduce rb_id_serial_t and rb_id_to_serial()
  to represent ID by serial number.
- internal.h: use id_table for method tables.
- class.c, gc.c, marshal.c, vm.c, vm_method.c: ditto.

Files

method_number.png	18.2 KB	08/06/2015	ko1 (Koichi Sasada)
measurement2.png	29.4 KB	08/11/2015	ko1 (Koichi Sasada)
microbenchmark.pdf	124 KB	08/11/2015	ko1 (Koichi Sasada)