Hi.

I think r43870 make benchmark/bm_so_k_nucleotide.rb slow.

r43870

% time ./miniruby ./benchmark/bm_so_k_nucleotide.rb
./miniruby ./benchmark/bm_so_k_nucleotide.rb  1.70s user 0.01s system 99% cpu 1.718 total

r43869

% time ./miniruby ./benchmark/bm_so_k_nucleotide.rb
./miniruby ./benchmark/bm_so_k_nucleotide.rb  1.52s user 0.03s system 99% cpu 1.559 total
Revision 44058 - 12/08/2013 01:52 AM - tmm1 (Aman Gupta)
hash.c: revert r43870 and add alternative parser patch for literal keys

- hash.c (hash_aset_str): revert r43870 due to performance issue [Bug #9188] [ruby-core:58730]
- parse.y (assoc): convert literal string hash keys to fstrings
- test/ruby/test_hash.rb (class TestHash): expand test

Revision 44058 - 12/08/2013 01:52 AM - tmm1 (Aman Gupta)
hash.c: revert r43870 and add alternative parser patch for literal keys

- hash.c (hash_aset_str): revert r43870 due to performance issue [Bug #9188] [ruby-core:58730]
- parse.y (assoc): convert literal string hash keys to fstrings
- test/ruby/test_hash.rb (class TestHash): expand test

Revision 1e83e15a - 10/26/2018 05:32 AM - normal
hash.c: aset deduplicates un-tainted string

We revisit [Bug #9188] since st.c is much improved since then, and benchmarks against so_k_nucleotide seem to indicate little or no performance change compared to before.
[ruby-core:89555] [Feature #15251]
From: Anmol Chopra chopraanmol1@gmail.com
git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@65371 b2dd03c8-39d4-4d8f-98ff-823f69b080e

Revision 65371 - 10/26/2018 05:32 AM - normalperson (Eric Wong)
hash.c: aset deduplicates un-tainted string

We revisit [Bug #9188] since st.c is much improved since then, and benchmarks against so_k_nucleotide seem to indicate little or no performance change compared to before.
[ruby-core:89555] [Feature #15251]
From: Anmol Chopra chopraanmol1@gmail.com

Revision 65371 - 10/26/2018 05:32 AM - normal
hash.c: aset deduplicates un-tainted string

We revisit [Bug #9188] since st.c is much improved since then, and benchmarks against so_k_nucleotide seem to indicate little or no performance change compared to before.
[ruby-core:89555] [Feature #15251]
From: Anmol Chopra chopraanmol1@gmail.com

History

#1 - 12/01/2013 06:05 PM - tmm1 (Aman Gupta)

=(

Maybe we should revert it?

#2 - 12/01/2013 07:23 PM - normalperson (Eric Wong)

Unfortunately, r43870 is a trade-off and optimized for frequently-reused keys.

How about only using rb_fstring for string literal hash keys?

#3 - 12/03/2013 08:20 AM - tmm1 (Aman Gupta)

How about only using rb_fstring for string literal hash keys?
I think this makes sense. We won't see a 10% savings like we did in #8998, but it's a fair compromise.

Do you have any thoughts on an implementation? I'll revert the existing commit in the meantime.

#4 - 12/03/2013 08:53 AM - normalperson (Eric Wong)
"tmm1 (Aman Gupta)" ruby@tmm1.net wrote:

Do you have any thoughts on an implementation? I'll revert the existing commit in the meantime.

Not really, but I think it needs to be done in the parser which I'm not familiar with. Hopefully you or somebody else has time/interest, I've got more critical problems/projects to attend to than performance.

#5 - 12/07/2013 11:57 AM - tmm1 (Aman Gupta)

There were some minor improvements to rb_fstring() recently, but the hash changes are still slow on trunk.

In my environment:

```
trunk
./miniruby -I. benchmark/bm_so_k_nucleotide.rb > /dev/null  3.72s user 0.03s system 99% cpu 3.757 total
./miniruby -I. benchmark/bm_so_k_nucleotide.rb > /dev/null  3.76s user 0.03s system 99% cpu 3.794 total
./miniruby -I. benchmark/bm_so_k_nucleotide.rb > /dev/null  3.70s user 0.03s system 99% cpu 3.736 total
./miniruby -I. benchmark/bm_so_k_nucleotide.rb > /dev/null  3.78s user 0.03s system 99% cpu 3.817 total
```

```
revert r43870
./miniruby -l benchmark/bm_so_k_nucleotide.rb > /dev/null  3.50s user 0.02s system 99% cpu 3.528 total
./miniruby -l benchmark/bm_so_k_nucleotide.rb > /dev/null  3.49s user 0.02s system 99% cpu 3.515 total
./miniruby -l benchmark/bm_so_k_nucleotide.rb > /dev/null  3.54s user 0.02s system 99% cpu 3.570 total
./miniruby -l benchmark/bm_so_k_nucleotide.rb > /dev/null  3.52s user 0.02s system 99% cpu 3.546 total
```

I'm trying to come up with an alternative patch to keep the string literal hash key test passing, but I'm having a hard time with the parser. The best I can come up with is the following, which works but feels wrong:

```ruby
@@ -535,7 +535,7 @@ newhash
 for (i = num; i > 0; i -= 2) {
   const VALUE v = TOPN(i - 2);
   const VALUE k = TOPN(i - 1);
-   rb_hash_aset(val, k, v);
+   rb_hash_aset(val, RB_TYPE_P(k, T_STRING) ? rb_fstring(k) : k, v); } POPN(num); }
```

Ideally, I'd like to replace alternate putstring instructions involved in any newhash instruction with putobject instructions. Could someone point out the best place in the compiler to do this?

#6 - 12/07/2013 05:32 PM - tmm1 (Aman Gupta)

After a few more false starts in the compiler, I ended up with the following patch to the parser.

```
diff --git a/parse.y b/parse.y
index 8207ad7..4629a60 100644
--- a/parse.y
+++ b/parse.y
@@ -4912,7 +4912,11 @@ assocs       : assoc
 assoc      : arg_value tASSOC arg_value
 {%
     $$ = list_append(NEW_LIST($1), $3);
-    if (nd_type($1) == NODE_STR) {
+    if (nd_type($1) == NODE_STR) {  
+      $$ = list_append(NEW_LIST(NEW_LIT(rb_fstring($1->nd_lit))), $3);
+    } else {
+      $$ = list_append(NEW_LIST($1), $3);
+    }
+  }%}

```
def test_ASET_string
  a = {"ABC" => :t}
  b = {"ABC" => :t}
  assert_same a.keys[0], b.keys[0]
  assert_same "ABC".freeze, a.keys[0]
end

def test_EQUAL # '=='
  #7 - 12/08/2013 10:52 AM - tmm1 (Aman Gupta)
  - Status changed from Open to Closed
  - % Done changed from 0 to 100

  This issue was solved with changeset r44058.
  Narihiro, thank you for reporting this issue.
  Your contribution to Ruby is greatly appreciated.
  May Ruby be with you.

  hash.c: revert r43870 and add alternative parser patch for literal keys
  - hash.c (hash_aset_str): revert r43870 due to performance issue [Bug #9188] [ruby-core:58730]
  - parse.y (assoc): convert literal string hash keys to fstrings
  - test/ruby/test_hash.rb (class TestHash): expand test

  #8 - 12/08/2013 10:56 AM - tmm1 (Aman Gupta)
  charliesome (Charlie Somerville) helped me with my compiler patch. It adds a new opt_aset_str instruction that handles the string literal hash_aset case (hsh["abc"]=).

  Unfortunately, it did not make a big difference in long-lived strings generated in our app. The patch follows for posterity.

diff --git a/compile.c b/compile.c
index 812f692..1cc038c 100644
--- a/compile.c
+++ b/compile.c
@@ -5290,6 +5290,24 @@ iseq_compile_each(rb_iseq_t *iseq, LINK_ANCHOR *ret, NODE * node, int poped)
               break;
 } else case NODE_ATTRASGN:
 - if (node->nd_mid == idASET &
 - node->nd_recv != (NODE *)1 &
 - node->nd_args &
 - nd_type(node->nd_args) == NODE_ARRAY &
 - node->nd_args->nd_alen == 2 &
 - node->nd_args->nd_head == NODE_STR)
 - {
 - COMPILE(ret, "recv", node->nd_recv);
 - COMPILER(ret, "value", node->nd_args->nd_next->nd_head);
  ADD_INSNS2(ret, line, opt_aset_str,
  new_callinfo(iseq, idASET, 2, 0, 0),
  rb_fstring(node->nd_args->nd_head->nd_lit));
  if (poped) {
  ADD_INSNS(ret, line, pop);
  }
  break;
 + } + DECL_ANCHOR(recv); DECL_ANCHOR(args); VALUE flag = 0; diff --git a/hash.c b/hash.c
index 1321b83..8bbd569 100644
--- a/hash.c
+++ b/hash.c
@@ -2368,7 +2368,7 @@ static VALUE rb_hash_compare_by_id_p(VALUE hash);
       h1["a"] #=> 100
       h1.compare_by_identity
h1.compare_by_identity? #=> true

* h1["a"] #=> nil # different objects.
* h1["a".dup] #=> nil # different objects.
* h1[:c] #=> "c" # same symbols are all same. */
diff --git a/insns.def b/insns.def
index 63a36b3..dcecfdc 100644 --- a/insns.def +++
b/insns.def @@ -1903,6 +1903,27 @@
@@ -897,7 +904,7 @@

end
def test_ASET_fstring_key
  a, b = {}, {}
  a["abc"] = 1
  b["abc"] = 1
  assert_same a.keys[0], b.keys[0]
end

def test_NEWHASH_fstring_key
  a = ["ABC" => 3]
  b = ["ABC" => 3]
  h = @cls[]
  h.compare_by_identity
  h["a"] = 1
  h["a"] = 2
  h["a".dup] = 2
  assert_equal(["a",1], h.assoc("a"))
end

Btw, I took some time to work on this further. Only very lightly tested (make check passes)

# GC::Profiler.report doesn't even show anything with this patch applied # because GC never happens.
"GC::Profiler.enable
10000000.times do
h["HI"] = 0
h["HI"]
GC::Profiler.report

Subject: [PATCH] prefreeze literal strings for hash aset/aref

Based on a patch by Charlie Somerville and Aman Gupta:

http://mid.gmane.org/redmine.journal-43505.20131208105635@ruby-lang.org

The following changes since commit 12b09864056bf9b961f06b0ef675b9fc2fabb9238:

* properties. (2014-01-03 01:51:05 +0000)

are available in the git repository at:

git://80x24.org/ruby.git opt_aref_aset_str

for you to fetch changes up to 2906ef4bf2aaa0873f198cc1a949c1cc7740be7f:

prefreeze literal strings for hash aset/aref (2014-01-03 03:44:51 +0000)

Eric Wong (1):
prefreeze literal strings for hash aset/aref

```
class Node
  attr_accessor :type, :mid, :recv, :args

def ndאירע(arr, len)
  if mid == idAREF &&
    recv
      if args &&
        args.array? &&
        type(args) == NODE_ARRAY &&
        len == 1 &&
        type(args.head) == NODE_STR
          VALUE str = rb_fstring(args.head.lit);
          args.head.lit = str;
          compile2(ret, recv, args, 1, 0, 0, str);
        end
      end
    end
  end
end
```

diff --git a/compile.c b/compile.c
index 5b28401..f12f40d 100644
--- a/compile.c
+++ b/compile.c
@@ -5300,6 +5317,25 @@ iseq_compile_each(rb_iseq_t *iseq, LINK_ANCHOR *ret, NODE * node, int poped)
          VALUE flag = 0;
          VALUE argc;
          if (node->nd_mid == idASET &&
```
node->nd_args->nd_alen == 2 &&
nd_type(node->nd_args->nd_head) == NODE_STR)
{
VALUE str = rb_fstring(node->nd_args->nd_head->nd_lit);
node->nd_args->nd_head->nd_lit = str;
COMPILE(ret, "recv", node->nd_recv);
COMPILE(ret, "value", node->nd_args->nd_next->nd_head);
ADD_INSN2(ret, line, opt_aset_str,
new_callinfo(iseq, idASET, 2, 0, 0), str);
if (poped) {
ADD_INSN(ret, line, pop);
}
break;
}

INIT_ANCHOR(recv);
INIT_ANCHOR(args);
argc = setup_args(iseq, args, node->nd_args, &flag);
差異 --git a/hash.c b/hash.c
index 0eca4b9..3cf7d8d 100644
--- a/hash.c
+++ b/hash.c
@@ -2390,7 +2390,7 @@ static VALUE rb_hash_compare_by_id_p(VALUE hash);

  h1["a"] #=> 100
  h1.compare_by_identity
  h1.compare_by_identity? #=> true

  h1["a"] #=> nil # different objects.

  h1["a.dup"] #=> nil # different objects.
  h1["c"] #=> "c" # same symbols are all same. * diff --git a/insns.def b/insns.def index ad4bba6..616838d 100644 --- a/insns.def +++
b/insns.def @@ -1903,6 +1903,47 @@
          b/insns.def @@ -1903,6 +1903,47 @@ opt_aset

 /**
  * @c optimize
  */
  @e recv{str = set
  @e  recv{str = set
* \+DEFINE_INSN +opt_aset_str +(CALL_INFO ci, VALUE key) +(VALUE recv, VALUE val) +(VALUE val) +{
* if ((SPECIAL_CONST_P(recv) && &RBASIC_CLASS(recv) == rb_cHash && &BASIC_OP_UNREDEFINED_P(BOP_ASET,
* HASH_REDEFINED_OP_FLAG)) { (BOP_ASET,
* rb_hash_aset(recv, key, val);}
* else {
  PUSH(recv);
  PUSH(rb_str_resurrect(key));
  PUSH(val);
  CALL_SIMPLE_METHOD(recv);
  } +} +*/
  @c optimize
  @e recv{str
  @e recv{str
* \+DEFINE_INSN +opt_aref_str +(CALL_INFO ci, VALUE key) +(VALUE recv, VALUE val) +(VALUE recv) +(VALUE val) +{
* if ((SPECIAL_CONST_P(recv) && &RBASIC_CLASS(recv) == rb_cHash && &BASIC_OP_UNREDEFINED_P(BOP_AREF,
* HASH_REDEFINED_OP_FLAG)) { (BOP_AREF,
* val = rb_hash_aref(recv, key);}
* else {
  PUSH(recv);
  PUSH(rb_str_resurrect(key));
  CALL_SIMPLE_METHOD(recv);
  } +} +*/
diff --git a/test/ruby/test_hash.rb b/test/ruby/test_hash.rb
index 70c0442..f5af4dd 100644
--- a/test/ruby/test_hash.rb
+++ b/test/ruby/test_hash.rb
@@ -209,6 +209,13 @@ class TestHash < Test::Unit::TestCase
    assert_equal(256, h[z])
end

def test_ASET_fstring_key
    a, b = {}, {}
    a["abc"] = 1
    b["abc"] = 1
    assert_same a.keys[0], b.keys[0]
end

+def test_NEWHASH_fstring_key
    a = {"ABC" => :t}
    b = {"ABC" => :t}
    h = @cls()
    h.compare_by_identity
    h["a"] = 1
    h["a"] = 2
    h["a.dup"] = 2
    assert_equal(["a",1], h.assoc("a"))
end

#10 - 01/06/2014 03:53 PM - ko1 (Koichi Sasada)
(2014/01/03 12:49), Eric Wong wrote:

    Btw, I took some time to work on this further. Only very lightly tested (make check passes)

    What the difference between charliesome/tmm1's patch?
    (just curious, I want to know)

    --
    // SASADA Koichi at atdot dot net

#11 - 01/06/2014 03:53 PM - normalperson (Eric Wong)
SASADA Koichi ko1@atdot.net wrote:

(2014/01/03 12:49), Eric Wong wrote:

    Btw, I took some time to work on this further. Only very lightly tested (make check passes)

    What the difference between charliesome/tmm1's patch?
    (just curious, I want to know)

    Mine replaces nd_lit in node directly (seems OK, since other rb_fstring uses do that). I also implement aref, not just aset.

    This is my first time working in compile.c and I'm not familiar with this code at all, but it still seems to be still working :)
    Other than my trivial loop, I haven't tested performance.
Mine replaces nd_lit in node directly (seems OK, since other rb_fstring uses do that). I also implement aref, not just aset.

This is my first time working in compile.c and I'm not familiar with this code at all, but it still seems to be still working :)
Other than my trivial loop, I haven't tested performance.

I got it.

My concern is performance regression with huge entries of fstring table with this technique. Maybe we can avoid such regression with smart data structure (for example, do not use st).

--
// SASADA Koichi at atdot dot net

I got it.

My concern is performance regression with huge entries of fstring table with this technique. Maybe we can avoid such regression with smart data structure (for example, do not use st).

Yes, st_table_entry is gigantic, I think we should go back to unordered st for things where order does not matter. But we can probably do better for the fstring table, even.

The new opt_aset_str and opt_aref_str instructions only affect string literals, and all strings literals are already in the fstring table in 2.1. I don't think there is any possible performance regression with this technique.

The design is based on st, but uses linked-list of cache-sized arrays for chaining, so it's as if each bucket is an st-packed array.

Fwiw, I don't think this is a good design for our internal data structures. I'll experiment with others in a few weeks/months where the lookup node is embedded in the struct itself. It would use offsetof (via container_of macro) to get back the original object, meaning we avoid extra pointer chasing. This won't be good for things which use VALUEs as keys, but probably good for method tables.

I'm not sure if we can/should change hash.c and its st.c usage, yet, due to public API compatibility.