accessing to instance variable should be fast.

04/12/2016 03:50 PM - tarui (Masaya Tarui)

Ruby master - Bug #12274

Status: Closed
Priority: Normal
Assignee:
Target version:
ruby -v: ruby 2.4.0dev (2016-04-12 trunk 54553) [x86_64-linux]
Backport: 2.1: UNKNOWN, 2.2: UNKNOWN, 2.3: UNKNOWN

Description
Currently, accessing to instance variable is quite slower than accessing to local variable. I think accessing to instance variable is basic operation and it should be fast, so tried to improve.

patch: https://github.com/tarui/ruby/commit/dd993da80c7ad84340689137bf8b308793595cae

On mame's optcarrot benchmark. (https://github.com/mame/optcarrot/) it is 10%(!) faster than trunk.
It increases in the maintenance cost a little, but can I commit it?

$ ./ruby --disable-gems ../../../optcarrot/bin/optcarrot --benchmark ../../../optcarrot/examples/Lan_Master.nes
ruby 2.4.0dev (2016-04-12 trunk 54553) [x86_64-linux]
fps: 13.664029283085743
checksum: 59662

$ ./ruby --disable-gems ../../../optcarrot/bin/optcarrot --benchmark ../../../optcarrot/examples/Lan_Master.nes
ruby 2.4.0dev (2016-04-12 fast-ivar-access 54553) [x86_64-linux]
fps: 15.120651593726231
checksum: 59662

Associated revisions

Revision 44916ec4 - 05/11/2016 12:50 PM - tarui (Masaya Tarui)

- compile.c (iseq_compile_each): share InlineCache during same instance variable accesses. Reducing memory consumption, raising cache hit rate and raising branch prediction hit rate are expected. A part of [Bug #12274].
  * iseq.h (struct iseq_compile_data): introduce instance variable IC table for sharing.
  * iseq.c (prepare_iseq_build, compile_data_free): construct/destruct above table.

Revision 54976 - 05/11/2016 12:50 PM - tarui (Masaya Tarui)

- compile.c (iseq_compile_each): share InlineCache during same instance variable accesses. Reducing memory consumption, raising cache hit rate and raising branch prediction hit rate are expected. A part of [Bug #12274].
  * iseq.h (struct iseq_compile_data): introduce instance variable IC table for sharing.
  * iseq.c (prepare_iseq_build, compile_data_free): construct/destruct above table.
compile.c (iseq_compile_each): share InlineCache during same instance variable accesses. Reducing memory consumption, raising cache hit rate and raising branch prediction hit rate are expected. A part of [Bug #12274].

iseq.h (struct iseq_compile_data): introduce instance variable IC table for sharing.
iseq.c (prepare_iseq_build, compile_data_free): construct/destruct above table.

iseq.h (struct iseq_compile_data): introduce instance variable IC table for sharing.
iseq.c (prepare_iseq_build, compile_data_free): construct/destruct above table.

iseq.h (struct iseq_compile_data): introduce instance variable IC table for sharing.
iseq.c (prepare_iseq_build, compile_data_free): construct/destruct above table.

vm_insnhelper.c (vm_getivar): describe fast-path explicit (compiler friendly). [Bug #12274].

evaluation result:
fps: 19.21335880758348
->
fps: 22.16285461090967

History
#1 - 04/12/2016 07:21 PM - ko1 (Koichi Sasada)
Tarui-san suggested another way to optimize and this is my version of that technique (with some refactoring).

fps: 19.21335880758348
->
fps: 22.16285461090967
vm_getivar(VALUE obj, ID id, IC ic, struct rb_call_cache *cc, int is_attr)
{
    #if USE_IC_FOR_IVAR
    if (RB_TYPE_P(obj, T_OBJECT)) {
        VALUE val = Qundef;
        VALUE klass = RBASIC(obj)->klass;
        VALUE ic_value;
        VALUE index;
        const long len = ROBJECT_NUMIV(obj);
        const VALUE *const ptr = ROBJECT_IVPTR(obj);
        if (LIKELY(is_attr ? cc->aux.index > 0 : ic->ic_serial == RCLASS_SERIAL(klass))) {
            long index = !is_attr ? (long)ic->ic_value.index : (long)(cc->aux.index - 1);
            if (LIKELY(is_attr ? cc->aux.index > 0 : ic->ic_serial == RCLASS_SERIAL(klass))) {
                long index = !is_attr ? (long)ic->ic_value.index : (long)(cc->aux.index - 1);
                if (index < len) {
                    val = ptr[index];
                } else {
                    st_data_t index;
                    struct st_table *iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
                    goto undefined;
                }
            } else if (RB_TYPE_P(obj, T_OBJECT)) {
                const long len = ROBJECT_NUMIV(obj);
                const VALUE *const ptr = ROBJECT_IVPTR(obj);
                st_data_t index;
                struct st_table *iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
                val = Qundef;
                if (iv_index_tbl) {
                    if (st_lookup(iv_index_tbl, id, &index)) {
                        if (is_attr) {
                            ic->ic_value.index = index;
                            ic->ic_serial = RCLASS_SERIAL(klass);
                        } else { /* call_info */
                            cc->aux.index = (int)index + 1;
                        }
                    } else if (RB_TYPE_P(obj, T_OBJECT)) {
                        const VALUE *const ptr = ROBJECT_IVPTR(obj);
                        struct st_table *iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
                        val = Qundef;
                        if (iv_index_tbl) {
                            if (st_lookup(iv_index_tbl, id, &index)) {
                                if (is_attr) {
                                    ic->ic_value.index = index;
                                    ic->ic_serial = RCLASS_SERIAL(klass);
                                } else { /* call_info */
                                    cc->aux.index = (int)index + 1;
                                }
                            } else if (is_attr) {
                                rb_warning("instance variable %s not initialized", QUOTE_ID(id));
                                val = Qnil;
                            } else { /* call_info */
                                cc->aux.index = (int)index + 1;
                            }
                    } else if (LIKELY(is_attr ? cc->aux.index > 0 : ic->ic_serial == RCLASS_SERIAL(klass))) {
                        const long len = ROBJECT_NUMIV(obj);
                        const VALUE *const ptr = ROBJECT_IVPTR(obj);
                        st_data_t index;
                        struct st_table *iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
                        long index = !is_attr ? (long)ic->ic_value.index : (long)(cc->aux.index - 1);
                        if (LIKELY(is_attr ? cc->aux.index > 0 : ic->ic_serial == RCLASS_SERIAL(klass))) {
                            long index = !is_attr ? (long)ic->ic_value.index : (long)(cc->aux.index - 1);
                            if (index < len) {
                                val = ptr[index];
                            } else {
                                st_data_t index;
                                struct st_table *iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
                                goto undefined;
                            }
                        } else {
                            st_data_t index;
                            struct st_table *iv_index_tbl = ROBJECT_IV_INDEX_TBL(obj);
                            goto undefined;
                        }
                    }
                if (UNLIKELY(val == Qundef)) {
                    if (!is_attr && RTEST(ruby_verbose))
                        rb_warning("instance variable %s not initialized", QUOTE_ID(id));
                    val = Qnil;
                    undefined:
                }
    }
}

03/20/2021
if (!is_attr && RTEST(ruby_verbose)) {
    rb_warning("instance variable \"PRIsVALUE\" not initialized", QUOTE_ID(id));
} -
return val;
+  return Qnil;
}
#endif /* USE_IC_FOR_IVAR */
if (is_attr)

Koichi Sasada wrote:

Tarui-san suggested another way to optimize and this is my version of that technique (with some refactoring).

The diff is hard to read, would you have a commit on GitHub or a patch file?

Tarui-san, could you explain a bit the technique?
I am not sure to understand, it seems vm_getinstancevariable already has some inline cache.
Is it some manual inlining in the instruction code + avoiding some ID2SYM/INT2FIX (but these two seem performed at compile time, so mostly irrelevant for the benchmark)?

there are 2 parts of optimization.

- share inline cache between same symbol(at compile.c)
- inline fast pass only and cut useless check(RB_TYPE_P).(at insns.def)

We can skip st_lookup from the 2nd insns by sharing cache.

Inlining register pass may have a bit penalty.
Cutting check was a accidental :-), but it is not necessary if cached serial equals class one.

It is not for avoiding ID2SYM (In fact, it is calculated every time :-), it is for sharing.
Please check the 0007 below

It has been updated by Benoit Daloze.
avoiding some ID2SYM/INT2FIX (but these two seem performed at compile time, so mostly irrelevant for the benchmark)?

Please check the 0007 below
there are 2 parts of optimization.

- share inline cache between same symbol (at compile.c)
- inline fast pass only and cut useless check(RB_TYPE_P). (at insns.def)

We can skip st_lookup from the 2nd insns by sharing cache.

Inlining register pass may have a bit penalty.

Cutting check was a accidental ;-), but it is not necessary if cached serial equals class one.

I see, thanks for explaining :)

About the object check, is it not problematic to do ((struct RBasic*)obj)->klass if obj is a tagged integer (since klass is the second member, after flags)?
Or is there a hidden check before doing that?

Thank you for pointing out.
I'll revive check.

About the object check, is it not problematic to do ((struct RBasic*)obj)->klass if obj is a tagged integer (since klass is the second member, after flags)?

Thank you for pointing out.
I'll revive check.

Applied in changeset r54976.

- compile.c (iseq_compile_each): share InlineCache during same instance variable accesses. Reducing memory consumption, raising cache hit rate and raising branch prediction hit rate are expected. A part of [Bug #12274].
  - iseq.h (struct iseq_compile_data): introduce instance variable IC table for sharing.
  - iseq.c (prepare_iseq_build, compile_data_free): construct/destruct above table.