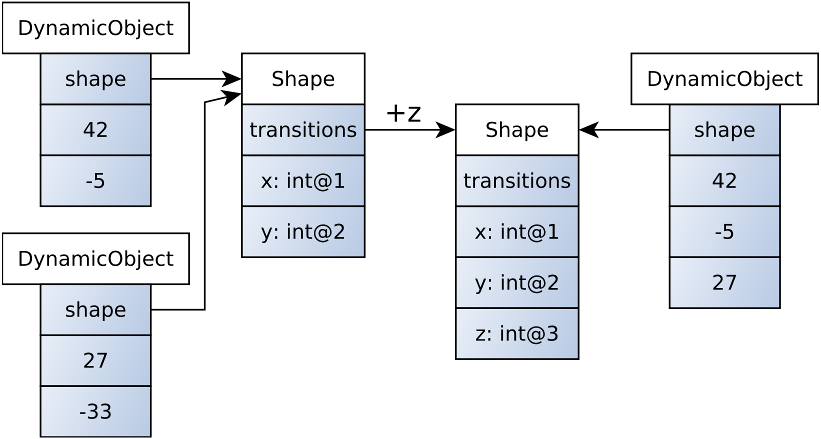
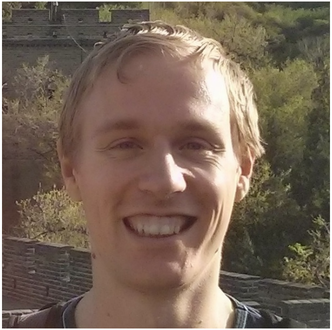
An efficient and thread-safe object representation for JRuby+Truffle

Benoit Daloze

Johannes Kepler University



Who am I?



Benoit Daloze

Twitter: @eregontp
GitHub: @eregon

- PhD student at Johannes Kepler University, Austria
- Research with JRuby+Truffle on concurrency
- Maintainer of the Ruby Spec Suite
- MRI and JRuby committer

How is it executed?

```
@ivar

@ivar = value
```

MRI 1.8

YARV

JRuby+Truffle

Summary in Ruby code

The Problem

One solution

Update on JRuby+Truffle

Conclusion

MRI 1.8: Finding '@' in the parser

```
// parse.y
yylex() {
    switch (character) {
      case '@':
        result = tIVAR;
    }
}

variable : tIVAR | ...

var_ref : variable
{
    node = gettable(variable);
}
```

MRI 1.8: In the Abstract Syntax Tree

```
// parse.y
NODE* gettable(ID id) {
    if (is_instance_id(id)) {
      return NEW_NODE(NODE_IVAR, id);
    }
    ...
}

// node.h
enum node_type {
  NODE_IVAR,
  ...
};
```

MRI 1.8: The interpreter execution loop

```c
// eval.c
VALUE rb_eval(VALUE self, NODE* node) {
  again:
    switch (nd_type(node)) {
      case NODE_IVAR:
        result = rb_ivar_get(self, node->nd_vid);
        break;
      ...
    }
}
```

MRI 1.8: Reading the variable from the object

```c
// variable.c
VALUE rb_ivar_get(VALUE obj, ID id) {
    VALUE val;
    switch (TYPE(obj)) {
      case T_OBJECT:
        if (st_lookup(ROBJECT(obj)->iv_tbl, id, &val))
            return val;
        break;
      ...
    }
    return Qnil;
}
```

MRI 1.8: The @ivar hash table

```c
// st.c
bool st_lookup(table, key, value) {
    int hash_val = do_hash(key, table);
    if (FIND_ENTRY(table, ptr, hash_val, bin_pos)) {
        *value = ptr->record;
        return true;
    }
    ...
}
```

YARV: In the bytecode compiler

```
// compile.c
int iseq_compile_each(rb_iseq_t* iseq, NODE* node) {
    switch (nd_type(node)) {
      case NODE_IVAR:
        ADD_INSN(getinstancevariable, node->var_id);
        break;
      ...
    }
}
```

YARV: Instruction definition

```
// insns.def
/**
  @c variable
  @e Get value of instance variable id of self.
 */
DEFINE_INSN
getinstancevariable
(ID id, IC ic)
()
(VALUE val)
{
    val = vm_getinstancevariable(GET_SELF(), id, ic);
}
```

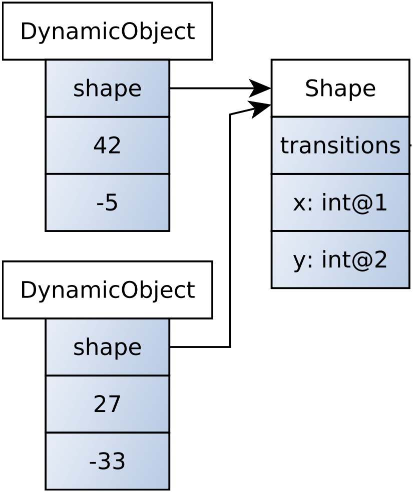
YARV: getinstancevariable fast path

```c
// vm_insnhelper.c
VALUE vm_getinstancevariable(VALUE obj, ID id, IC ic) {
    if (RB_TYPE_P(obj, T_OBJECT)) {
        VALUE klass = RBASIC(obj)->klass;
        int    len = ROBJECT_NUMIV(obj);
        VALUE* ptr = ROBJECT_IVPTR(obj);

        if (LIKELY(ic->serial == RCLASS_SERIAL(klass))) {
            int index = ic->index;
            if (index < len) {
                return ptr[index];
            }
        }
```

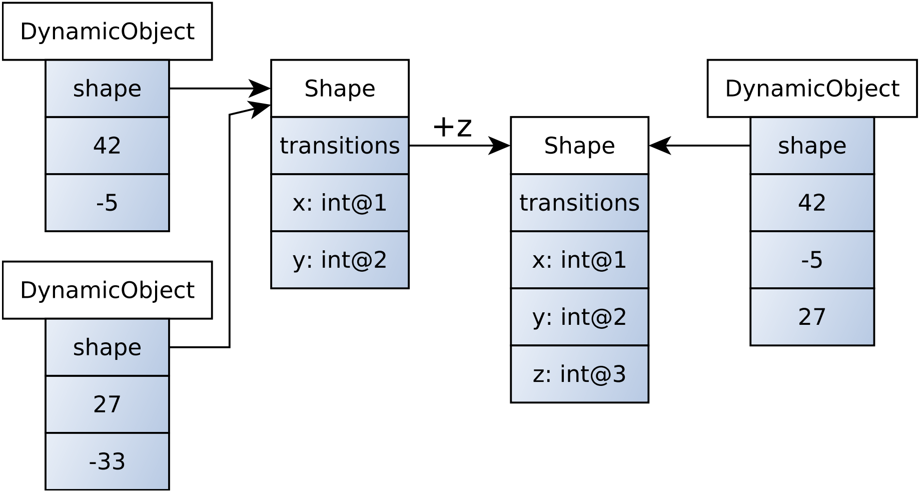
YARV: getinstancevariable slow path

```
else {
    st_data_t index;
    st_table *iv_index_tbl =
          ROBJECT_IV_INDEX_TBL(obj);

    if (st_lookup(iv_index_tbl, id, &index)) {
        ic->index = index;
        ic->serial = RCLASS_SERIAL(klass);
        if (index < len) {
            return ptr[index];
        }
    }
}
...
```

The Truffle Object Storage Model



The Truffle Object Storage Model



An Object Storage Model for the Truffle Language Implementation Framework

Reading an @ivar in JRuby+Truffle

```
class ReadInstanceVariableNode extends Node {
  final String name;

  @Specialization(guards = "object.getShape() == shape")
  Object read(DynamicObject object,
    @Cached("object.getShape()") Shape shape,
    @Cached("shape.getProperty(name)") Property property) {
      return property.get(object);
  }
}
```

MRI 1.8

```
table = obj.ivar_table
h = table.type.hash(id)
i = h % table.num_bins
entry = table.bins[i]
if entry.hash == h and table.type.equal(entry.key, id)
  return entry.value
end
```

YARV

```
if obj.klass.serial == cache.serial
  if obj.embed? and cache.index < 3
    return obj[cache.index]
  end
end
```

JRuby

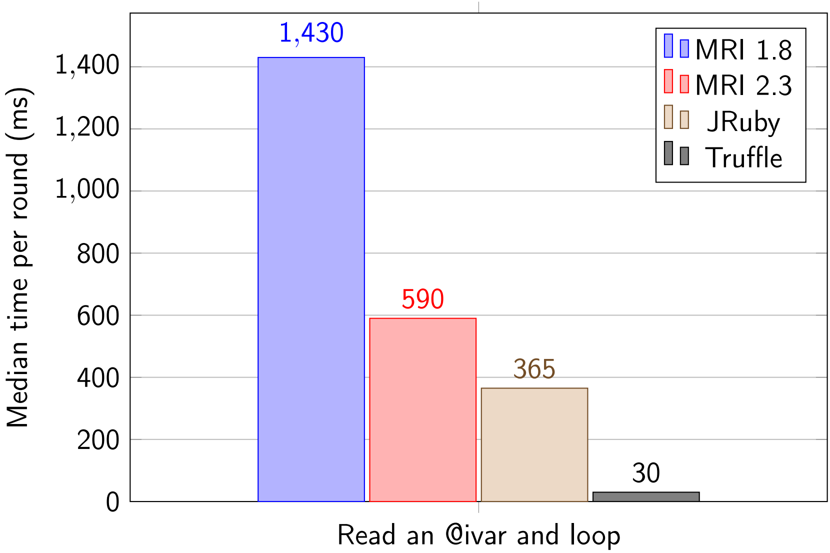
```
if obj.metaclass.realclass.id == CACHED_ID
  if CACHED_INDEX < obj.ivars.length
    return obj.ivars[CACHED_INDEX]
  end
end
```

```
if obj.shape == CACHED_SHAPE
  return obj[CACHED_INDEX]
end
```

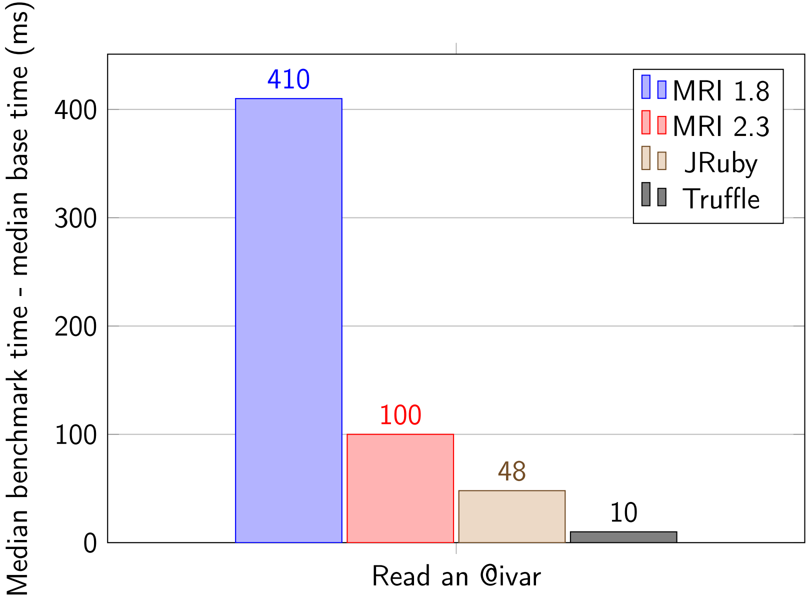
Simple benchmark: Read an @ivar

```ruby
class MyObject
  attr_reader :ivar
  def initialize
    @ivar = 1
  end
end

100.times {
  s = 0
  obj = MyObject.new
  puts Benchmark.measure {
    10_000_000.times {
      s += obj.ivar
    }
  }
}
```

Comparison: Read an @ivar



Comparison: Read an @ivar (time of benchmark - base)



The problem with concurrently growing objects

- Ruby objects can have a dynamic number of instance variables
- The only way to handle that is to have a growing storage
  - Or have a huge storage (Object[100] ?) but it would waste memory, limit the numbers of ivars, introduce more pressure on GC, etc.
- The underlying storage is always some chunk of memory.
- A chunk of memory cannot always grow in-place (`realloc` may change memory addresses)

The problem with concurrently growing objects

- Copying and changing a reference to this chunk cannot be done atomically, unless some synchronization is used

Consequences:

- Updates concurrent to definition of ivars might be lost
- Concurrent definition might lose ivars entirely
- Both are forbidden by the proposed Memory Model for Ruby https://bugs.ruby-lang.org/issues/12020

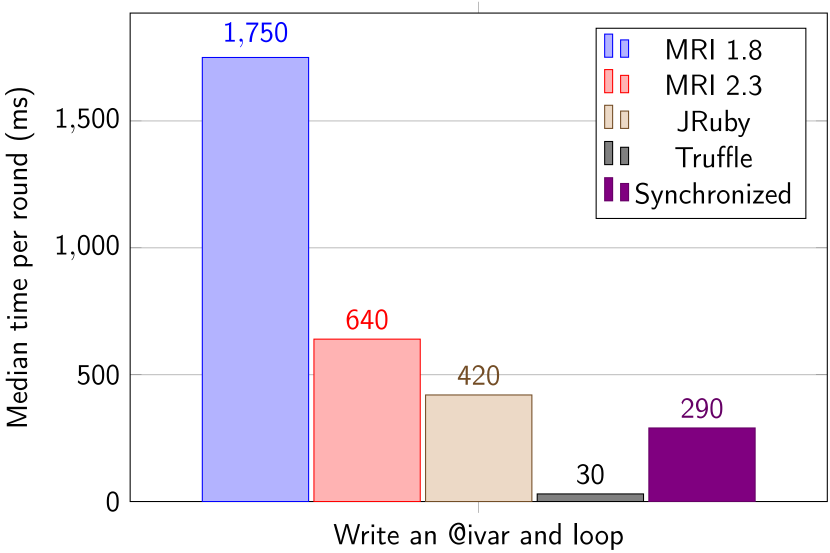
Is there a simple synchronization fix ?

```
def ivar_set(obj, name, value)
  obj.synchronize do
    if obj.shape == CACHED_SHAPE
      obj.ivars[CACHED_INDEX] = value
    end
  end
end

def new_ivar(obj, name, value)
  obj.synchronize do
    if obj.shape == OLD_SHAPE
      obj.shape = NEW_SHAPE
      obj.grow_storage if needed?
      obj.ivars[CACHED_INDEX] = new_value
    end
  end
end
```

Simple benchmark: Write an @ivar

```ruby
class MyObject
  attr_writer :ivar
  def initialize
    @ivar = 0
  end
end

100.times {
  s = 0
  obj = MyObject.new
  puts Benchmark.measure {
    10_000_000.times {
      s += 1
      obj.ivar = s
    }
  }
}
```

Comparison: Write an @ivar



My experiment in JRuby+Truffle

The idea:

- Only synchronize on globally-reachable objects
- All globally-reachable objects are initially *shared*, transitively
- Writing to a shared object makes the value shared as well

Sharing the roots: Statistics

2352 objects shared when starting a second thread:

681	Class
651	String
340	Symbol
101	Encoding
53	Module
15	Array
11	Hash
6	Proc
4	Object, Regexp
3	File, Bignum
2	Mutex, Thread
1	NilClass, Complex, Binding

Optimizations

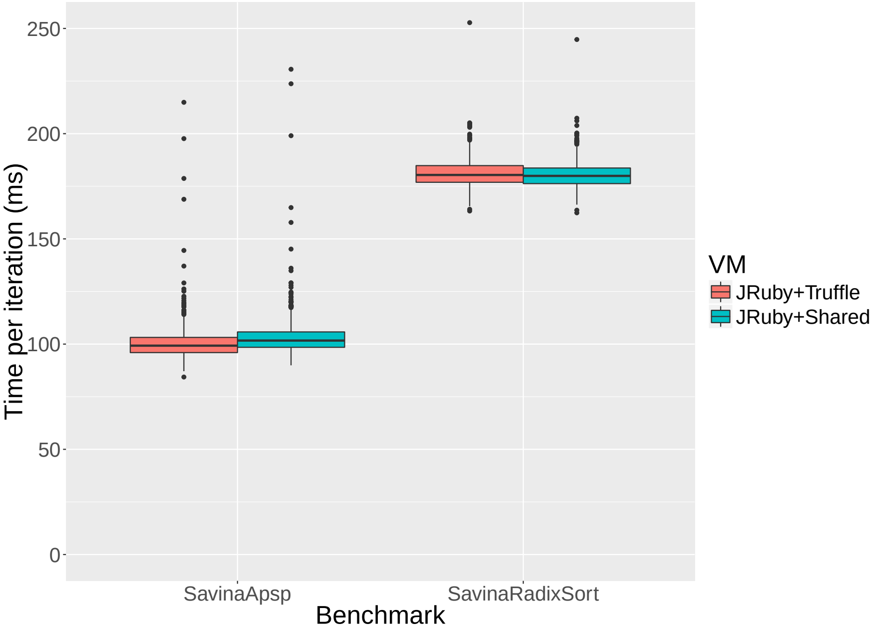
- The *shared* flag is part of the Shape
- So we can specialize on *shared* and *local* objects
- No overhead for *local* objects
- Setting the *shared* flag of one object is
  `obj.shape = SHARED_SHAPE`

Sharing the new value and its references

- Solution: specialize on the value structure

```ruby
# Nothing to share
obj.ivar = 1

# Share an Object
obj.ivar = Object.new

# Share an Array, an Object, a Hash and two Symbols
obj.ivar = [Object.new, { a: 1, b: 2 }]
```

Performance on 2 actor benchmarks from the Savina suite



MRI 1.8

YARV

JRuby+Truffle

Summary in Ruby code

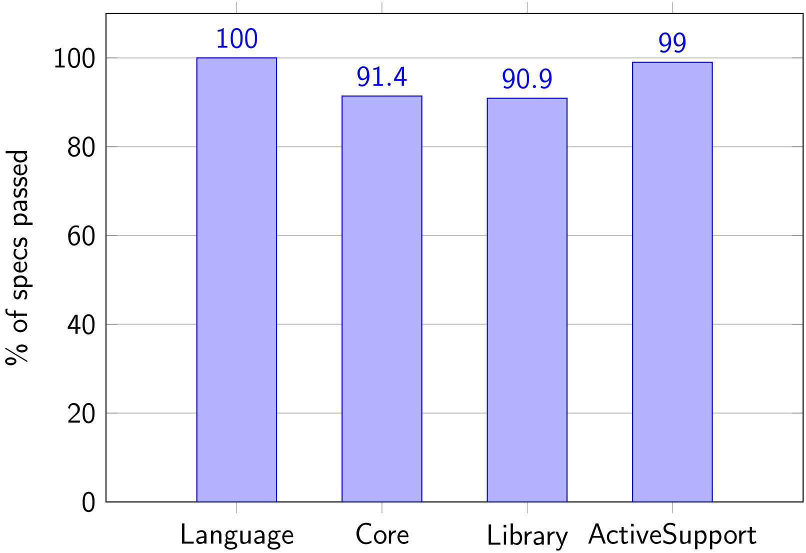
The Problem

One solution

Update on JRuby+Truffle

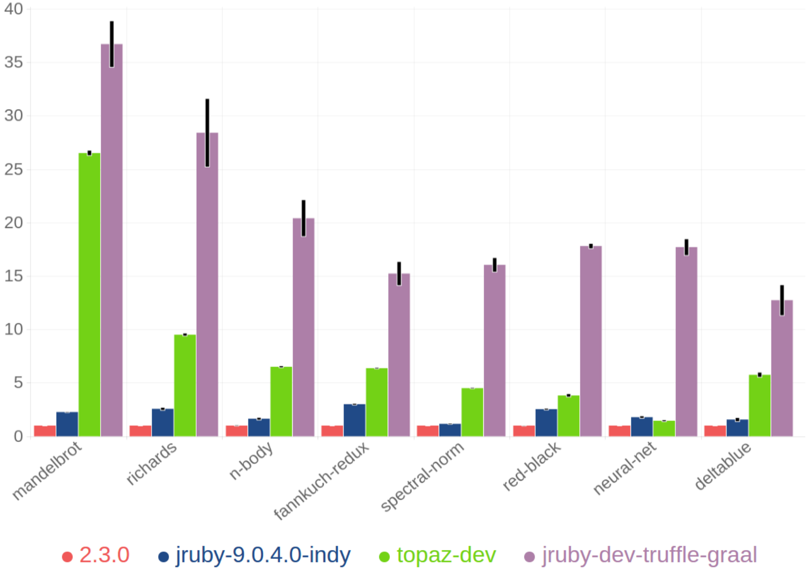
Conclusion

Compatibility



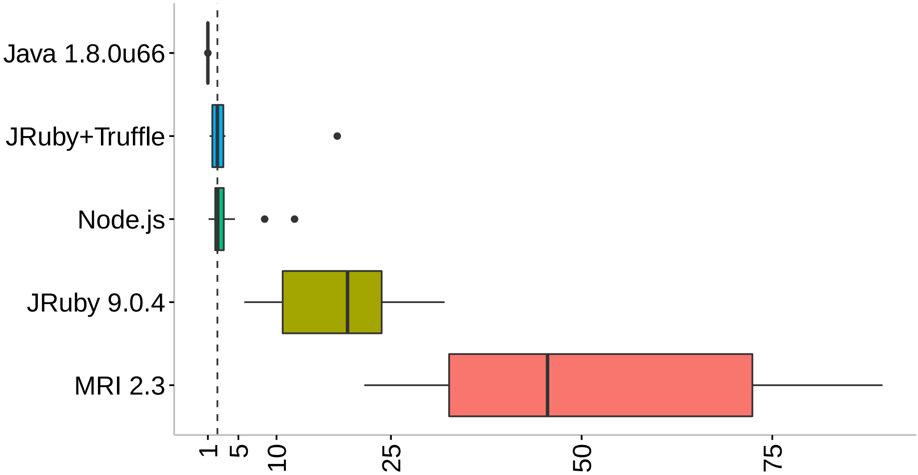
Based on the Ruby Spec Suite https://github.com/ruby/spec

Performance: Speedup relative to MRI 2.3



http://jruby.org/bench9000/

Performance: Are we fast yet?



https://github.com/smarr/are-we-fast-yet

Conclusion

- Concurrently growing objects need synchronization to not lose updates or new ivars
- This synchronization can have low overhead if we focus on what is actually needed
- JRuby+Truffle is a very promising Ruby implementation