software pilots TRIFORK.

Ruby on the JVM

Kresten Krab Thorup, Ph.D.
CTO, Trifork

A bit of history...

Smalltalk → OTI Smalltalk

Self

Strongtalk

VisualAge

HotSpot
Sun

IBM Java
IBM

Adaptive Optimizations

- **Key insight:** The VM knows more about your program than you do.
- **Consequence:** Let the VM adapt to program's behavior
  - VM will observe, tally and measure
  - feed information into successive optimizations

Time/Space Trade Off

- **Classical compiler "ideology"**
  - "ahead of time" compilers don't know which parts of the code to optimize
  - gcc -O0 ... -O6
- **Adaptive VMs**
  - Affords letting the program run for a while to see where optimizations will pay off.

The Ruby Nature

- **Program is created as it is being executed**
  - Class / module declarations are really statements, not declarations.
  - Programming style employs meta programming extensively
- **Very similar to Java, just "worse" :-)**

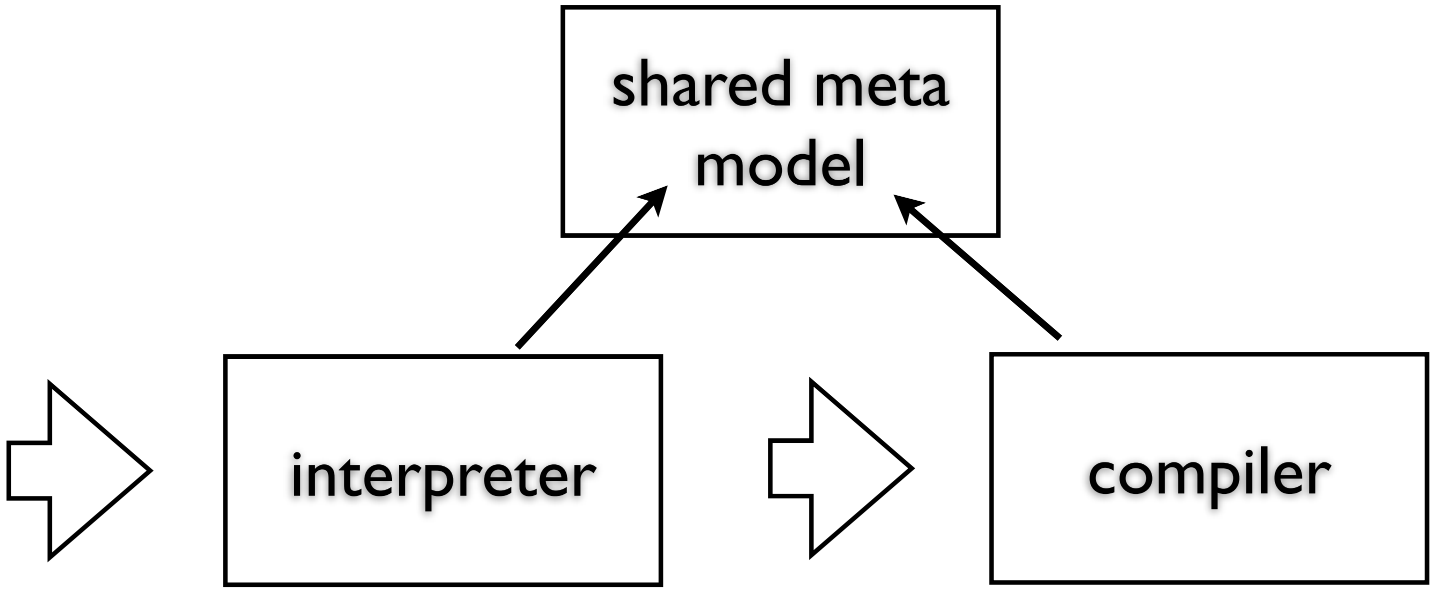
“Just In Time” VMs

- For interpreted-style languages, perform compilation when the program definition is known.
- AFAIK Strongtalk/HotSpot brought the innovation of a two-level VM:
  - start interpreted (running byte code)
  - optimize adaptively

The “HotRuby” project

- **Explore a “Server VM” for Ruby based on Java**
- Assume “long running processes” where we can afford “slow start”.
- Assume aggressive memory usage
- Exploit knowledge of how the JVM optimizes programs

HotRuby Architecture



~MRI performance ~2.5 x YARV performance

Design Philosophy

- Develop compiler and interpreter in parallel, and
- Favor compiler in the design of the runtime meta model
- Make trade-offs that reduce memory usage
- Write as much as possible in Ruby itself

Major Head Aches

- **Method invocation**
  - Calling "virtual" methods is slow
  - Program can change in many ways while running
- **Memory management**
  - Garbage collection is a resource hog

Naive Implementation

```
class RubyObject {
   RubyClass isa;
   HashTable<String,RubyObject> ivars;
   boolean frozen, tainted;
}
```

Naive Implementation

```
class RubyModule extends RubyObject {
  RubyVM vm;
  List<RubyModule> included_modules;
  HashTable<String,Callable> imethods;
  HashTable<String,Callable> mmethods;
  HashTable<String,RubyObject> constants;
}

class RubyClass extends RubyModule {
  RubyClass super_class;
}
```

Naive Implementation

```
class Callable {
   RubyObject call(RubyObject self,
                   RubyObject[] args,
                   RubyBlock block,
                   CallContext ctx);
}
```

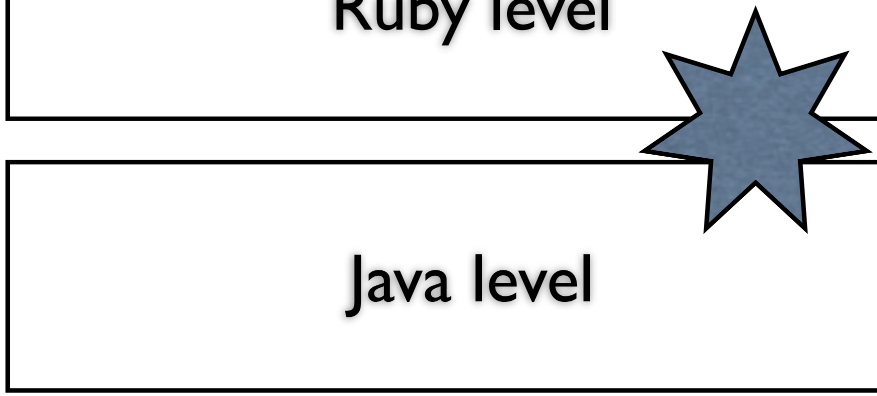
Naive Implementation

```
def m(obj)
  obj.foo(1, BAR)
end
```

... translates into something like ...

```
ctx = new MethodActivation(...);
ctx.set_local(0, args[0]);
obj = ctx.get_local(0)
one = ctx.new_fixnum(1);
bar = ctx.lookup_const("BAR");
callable = obj.isa.imethods.get("foo");
callable.call(obj, [one, bar], null, ctx)
```

Naive Implementation



Optimizing Calls

- Special-case common method names for core classes (`new`, `+`, `-`, `[]`, ...): They turn into Java-level virtual calls.
- Compiled code is "specialized", ...
- Method lookup is "compiled", ...

Method Specialization

- Compiled code is “specialized” for the receiving type,
    - making self-calls non virtual,
    - reducing public/private/protected checks: Security-checks happen at method-lookup, not invocation time.
    - making constant lookups really constant.

Compiled Lookup

- With the "Naive" implementation, method lookup is data-driven (HashTable).
- Compiled lookup means that we dynamically generate/modify code, as the lookup table changes.
- Allows the JVM's optimizing compiler to "know" how/when to eliminate or inline lookups.

Reduce Footprint

- Reduce size of heap for “active state” in a virtual machine
- Reduce “object churn”, i.e. rate of generated garbage.

Reducing Footprint

- Java objects already have an "isa" pointer! The implicit class reference.
- Use Java-level instance variables (in most cases)
- Eliminate the arguments array for method invocations (in most cases).
- Use Java-level local variables, removing the need for a "MethodActivation" object for each method call.

HotRuby Object

```
class RubyFoo {
   ObjectState state = null;
   RubyClass isa()
      { return state==null
          ? RubyClassFoo.class_object
          : state.singletonClass; }
}

class ObjectState {
    boolean frozen, tained;
    RubyClass singletonClass;
    HashTable<String,RubyObject> ivars;
}
```

HotRuby @ivars

- Generate Java classes lazily, upon first instantiation.
- At that point, analyze all applicable methods for reference to @ivars
- Generate Java-level ivars for all such references.
- Additional ivars go into ObjectState's hash table.

Reducing Footprint

- The "Naive" implementation has an overhead per object of
  20 bytes + ~20 bytes / ivar
- HotRuby ideally reduces this to
  12 bytes + 4 bytes / ivar
- Heap of 100.000 object with an average 3 ivars => 83% memory saving.

Use Java-Level locals

- The "cost" for having MethodActivation objects is both
  - The memory it consumes
  - The fact that such memory needs to be garbage collected
- Fall-back to MethodActivation object strategy for methods that call `eval` (and friends), and for local variables referenced from inner blocks.

HotRuby Status

- Runs basic Ruby programs (most importantly runit)
  - No Continuations, ObjectSpace, debugger, ... and many core classes
- Performance at 2.5 x YARV
- No, it does not run Rails.

software pilots TRIFORK.

Thanks