

Towards a Universal VM

The Java VM is not just for Java anymore

Alex Buckley
Spec lead, Java Language & VM
Sun Microsystems

Overview

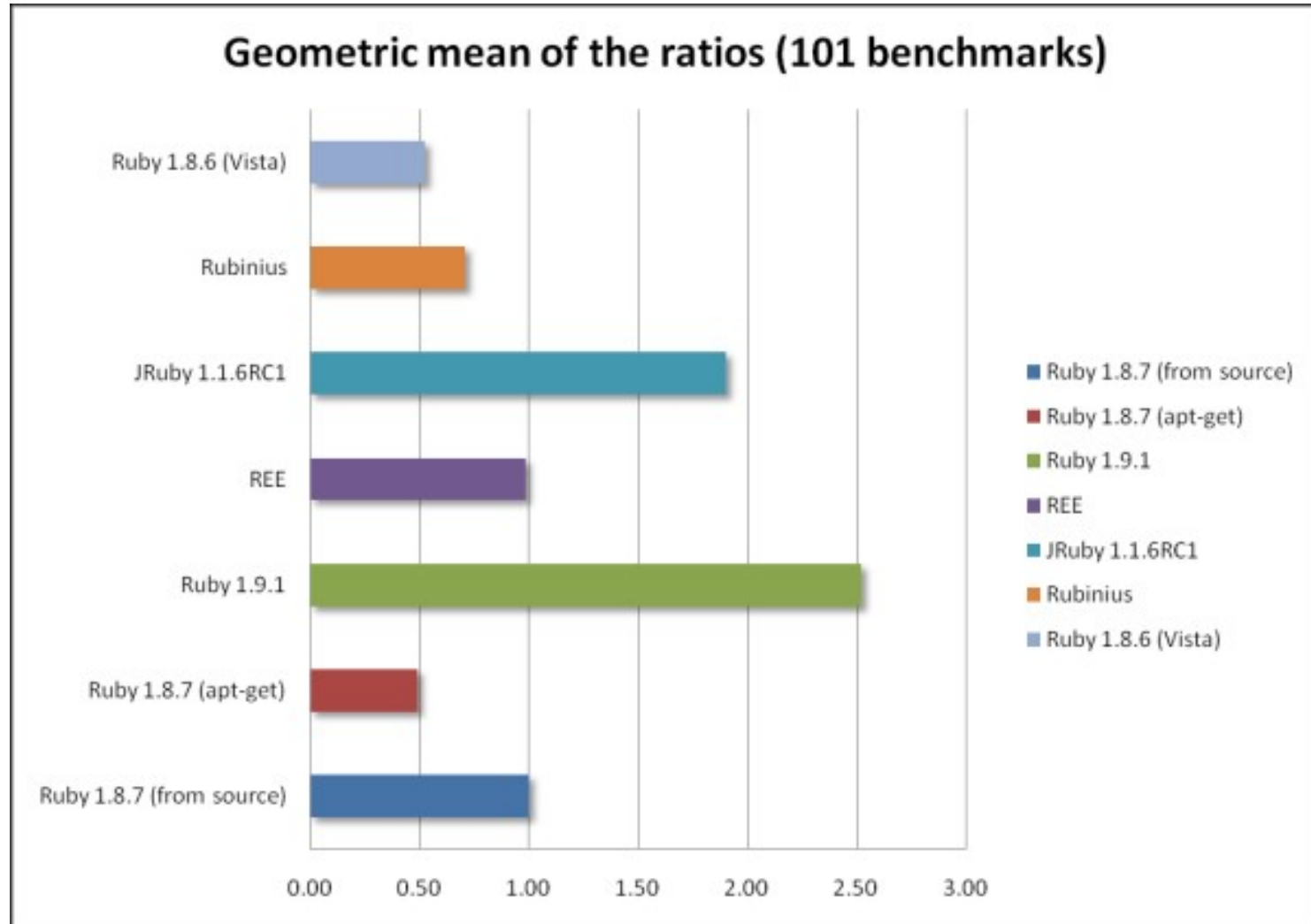
The Java VM has, in large part, been the engine behind the success of the Java language.

In years to come, the Java VM will power the success of other languages too.

JVM Specification, First Edition (1997)

- "The Java Virtual Machine knows nothing about the Java programming language, only of a particular binary format, the class file format."
- "A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information."
- "Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine."
- "Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages."

The Great Ruby Shootout 2008



<http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-december-2008/>

Java VM vs Java Language

Java language
fictions

Ruby language
fictions

Java VM
features

Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode

Checked exceptions
Generics
Enums
Overloading
Constructor chaining
Program analysis
Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Unicode

Open classes
Dynamic typing
'eval'
Closures
Mixins
Regular expressions
~~**Primitive types+ops**~~
Object model
Memory model
Dynamic linking
~~**Access control**~~
GC
~~**Unicode**~~

Towards a Universal VM

Easy to implement language fictions in a source compiler

- Checked exceptions in Java

- Traits and views in Scala

Some are very difficult to implement **efficiently**

- Open classes in Ruby

- Alternate numeric towers a la Scheme

Biggest mismatch between JVM and dynamic languages is in method selection and invocation

- Calling a method is cheap; selecting the right method is expensive

- Static languages do most of their method selection at compile-time

- Dynamic languages do almost none at compile-time (obviously)

Virtual method invocation in Java

javac compiles:

```
String s = "Hello World";  
System.out.println(s);
```

to:

```
ldc #2          // String "Hello World"  
astore_1       // Store in local var #1  
getstatic #3   // Field java/lang/System.out:  
               Ljava/io/PrintStream;  
aload_1        // Load string ref from local var #1  
invokevirtual #4 // Method  
               java/io/PrintStream.println:  
               (Ljava/lang/String;)V
```

Important! javac has statically chosen a version of println()

Virtual method invocation in the JVM

Given

```
invokevirtual Foo.bar:(int)int
```

the JVM looks for a method with signature

```
bar:(int)int
```

in the class of the receiver

(Verifier already proved that the receiver is `Foo` or a subtype)

If the receiver's class doesn't have that method, the JVM
recurses to its superclass, all the way up to `Foo`

Repeated recursive method lookup makes invocation slow

Fortunately, this can often be heavily optimized

Optimized virtual method invocation

Devirtualize monomorphic methods

Prove there is only one target method and invocation reduces to jump!
Can then inline the method call and get a bigger basic block
New kinds of optimizations then become possible

Inline caching

Figure out the most likely *dynamic* receiver type for a call site
Cache it by JIT-compiling the call site as follows:
 "If the dynamic receiver is of class X, jump to Y,
 otherwise do a normal (slow) virtual dispatch"
Optimizes for the common case

These techniques apply to all classfiles, not just those from javac

Dynamically typed method invocation

Compiling dynamic languages directly to the JVM is tricky

Consider this statically untyped code:

```
function max(x, y) {  
    if x.lessThan(y) then y else x  
}
```

What do we compile `x.lessThan(y)` to?

```
invokevirtual lessThan: (unknownArgType) boolean
```

No receiver type :-)

No static argument type :-)

Invalid instruction :-)

Dynamically typed method invocation

Perhaps we could pretend the code is statically typed:

```
boolean max(Object x, Object y) {  
    if (x.lessThan(y)) return y; else return x;  
}
```

Then compilation is straightforward:

```
invokevirtual Object.lessThan: (Object)boolean
```

Object does not have a lessThan() method :-)

NoSuchMethodError :-)

(Verifier will also get upset if you apply primitive ops to Objects)

Simulating dynamically typed invocation

Define an interface for each possible method signature

Complex, fragile, expensive

Use reflection everywhere

Use inline caching to invoke `java.lang.reflect.Method` objects for specific combinations of receiver+argument types

Heavyweight and slow if you use it for every method call

(Despite impressive improvements in reflective performance)

Always seem to be in a static typing straitjacket!

Perhaps the JVM isn't really a match for dynamic languages...

JVM Specification, First Edition (1997)

- "The Java Virtual Machine knows nothing about the Java programming language, only of a particular binary format, the class file format."
- "A class file contains Java Virtual Machine instructions (or bytecodes) and a symbol table, as well as other ancillary information."
- "Any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine."
- "Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages."
- "In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages."**

A little help goes a long way...

Static typing in the JVM is closer to the surface than it appears

Sun's JVM implementation is internally similar to Smalltalk

Smalltalk / Strongtalk is dynamic *and* fast

Need to fool the verifier to get past static type checking

The big need: *first-class language-specific method resolution*

So a language runtime can compute the real target of a call site

And then *get out of the VM's way*

This is the rationale behind **invokedynamic**

VM up-calls into language runtime to resolve method

Language runtime decides whether or not it needs to stay in the loop

invokedynamic

Consider this (hypothetical) instruction:

```
invokedynamic Object.lessThan: (Object) boolean
```

Suppose the receiver is an Integer and the argument is a Long

We know that invokedynamic should act as if it were:

```
invokevirtual Integer.lessThan: (Long) boolean
```

If the VM knew that, it could find the method *and inline it*

The language runtime can bridge the gap, if we ask it nicely

Method selection in dynamic languages

Ideally, a language runtime would see:

```
invokedynamic Object.lessThan: (Object) boolean
```

and do what javac does with static types at compile-time:

- Inspect the dynamic type of the receiver

- Inspect the dynamic type of the argument

- Check which methods are available *now* in the receiver's dynamic type

- Decide if a single argument is acceptable to those methods based on language rules for arity, optional parameters, default parameters...

- Box values of primitive type to values of reference types

- Box the argument list into an array and optimize it

and ask the VM to make this invocation happen:

```
invokevirtual Integer.lessThan: (Long) boolean
```

A language runtime wants to do all this...

ONCE

(Until a different object is assigned to the receiver variable,
or the receiver object's dynamic type is changed,
or the argument object's dynamic types are changed)

Bootstrap methods

The first time the JVM executes:

```
invokedynamic Object.lessThan: (Object)boolean
```

It calls a bootstrap method which does all those things

Bootstrap chooses the ultimate method to be called

```
Integer.lessThan: (Long)boolean
```

VM associates that method with this invokedynamic instruction

The next time the JVM executes:

```
invokedynamic Object.lessThan: (Object)boolean
```

It jumps to the previously chosen method immediately

No language runtime involved!

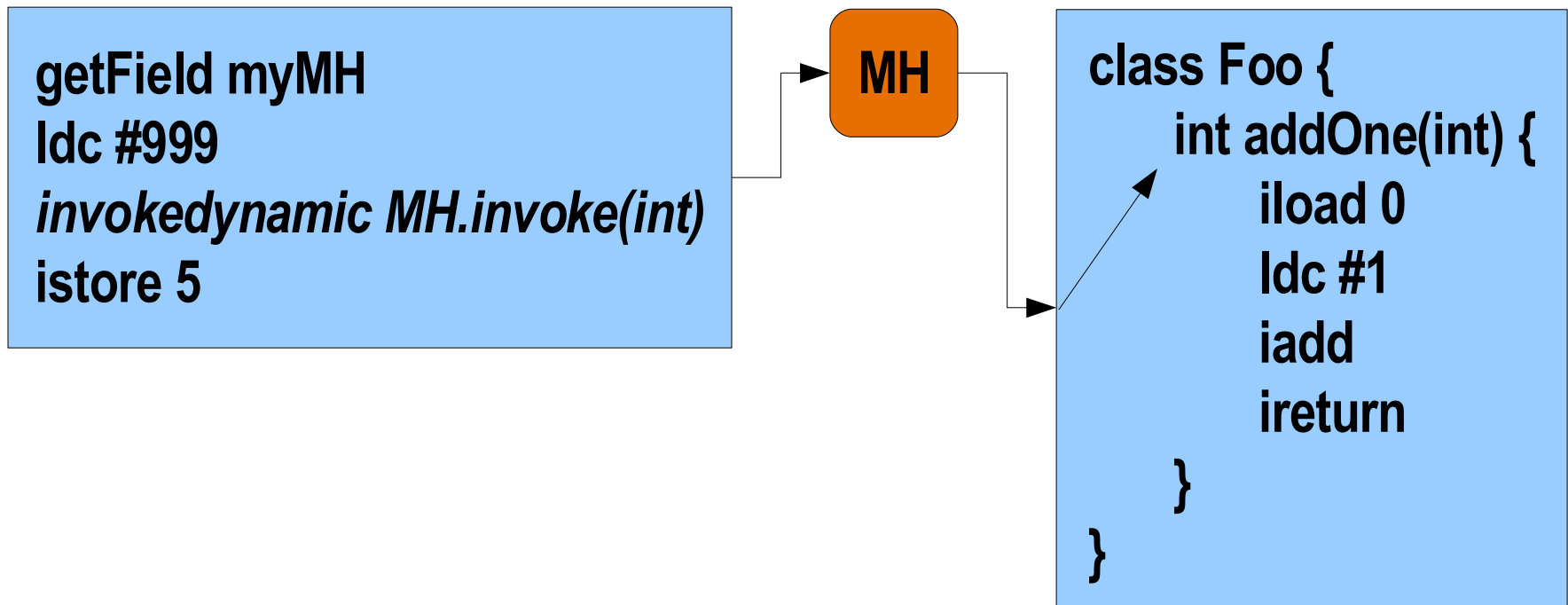
Hotspot's inliner can go to town

"We're basically a direct participant in the JVM's method selection and linking process. So cool."
- Charles O. Nutter

Method handles

A method handle is an object-oriented function pointer

Represents the bootstrap method's desired target method



Putting it all together

JVM method invocation is still statically typed

Dynamic typing hidden behind MethodHandle

Language runtime manages MethodHandles as it wishes

- Depending on the language's rules

- Can install/uninstall for a call site frequently or infrequently

- Can invoke a method of different name than in the instruction

- Can silently convert types, e.g. non-Unicode (Ruby) to Unicode (JVM)

- Can pre-allocate stack frames and thread-local context

- Can simulate currying and partial evaluation

- And so on!

"Bit by bit, piece by piece, the complex vagaries of our call protocols can be decomposed into functions, referenced by method handles, and composed into fast, efficient, direct calls"
- Charles O. Nutter

Tail calls

Idea

If A calls B calls C,

and the last action of B is to call C,

then the stack is adjusted as if A had directly called C

If A calls itself recursively, only one stack frame results

No more StackOverflowError!

Notes

"last action" is idiomatic - B *could* tail call C if B's code after the tail call uses no formal parameters, but that's hard to prove

Soft tail call: runtime *may* detect and implement tail call

Hard tail call: runtime *must* detect and implement tail call

Tail calls

wide instruction modifies the behavior of the next instruction

Generally used to access extended (2^{16}) local vars

Arnold Schwaighofer has used it to denote *hard* tail call when placed before an `invokevirtual`

Verification

`invokevirtual` is immediately followed by a return instruction

No exception handlers apply to the `invoke` instruction

The caller method is not **synchronized** and is holding no locks

(Has executed enough `monitorexit`'s to undo `monitorenter`'s)

Caller method's return type is identical to the callee method's

Tail calls in the Java security architecture

Can implement most tail calls by removing caller's stack frame

But *some* frames must not be removed

Access control

Every class is associated with a *protection domain (PD)*

PD represents a set of permissions for a given code source

e.g. FilePermission, SocketPermission, ReflectPermission

As methods invoke each other, their classes' PDs are tracked

On access to a guarded resource, permissions are computed by intersecting all protection domains on the stack

"Understanding Java Stack Inspection", Wallach and Felten, 1998

"A Tail-Recursive Machine with Stack Inspection", Clemens and Felleisen, 2004

Access control

Stack inspection logically requires a call crossing a PD to preserve the caller's PD and callee's PD

Degenerate case: *every* call is to a method in a different PD

Implications for tail call mechanism

Approach 1: Tail call only if caller domain == callee domain

Conservative, Easy to implement

Approach 2: Tail call if caller domain *or a domain already on the stack* == callee domain

Inspects the stack on **every** method call, not just security-sensitive calls

Approach 2b (Arnold): When a tail call would cross a PD, compress the stack by removing frames in the same PD

JSR 292

`invokedynamic` instruction

Method handles

Tail calls?

Interface injection?

Tuples?

Value objects?

Continuations?

Resources

John Rose (JSR 292 spec lead)

<http://blogs.sun.com/jrose/>

Charles O. Nutter (JRuby lead)

<http://blog.headius.com/>

Multi-Language Virtual Machine OpenJDK project

<http://openjdk.java.net/projects/mlvm/>

JVM Language Summit, September 2008

<http://openjdk.java.net/projects/mlvm/jvmlangsummit/>

"JVM Languages" Google Group

<http://groups.google.com/group/jvm-languages/>

