New Languages on the JVM:
Pain Points and Remedies

January 28, 2008

John R. Rose, Sr. Staff Engineer
john.rose@sun.com
http://blogs.sun.com/jrose

Agenda

- Opportunities
- Problems
- Case studies
- Solutions
- Ruby and the JVM
- *(Your item here...)*

Opportunities...

- VM-based systems have become normal
- CPU cycles are cheap enough for JIT, GC, RTT, ...
- many Java programmers, tools, systems
- much of the ecosystem is now open-source

Great (J)VM features

- flexible online code loading (with nice safe bytecodes)
- GC & object schema
- reflective access to classes & objects
- lots of ancillary tools (JMM, JVMTI, dtrace)
- good libraries & a nice language to write more
- optimizing JIT, object- and library-aware
- clever performance techniques:
  - type inference, customization, profiling, deoptimization, fast/slow paths, etc., etc.

Opportunities...

Bottom line...

VMs and tools are both mature and ubiquitous

- So what shall we build now...?
  - > partial answer: more languages!
- There seem to be about 200 JVM language implems:
  `http://robert-tolksdorf.de/vmlanguages.html`

Opportunities...

High level languages often require:

- very late binding (runtime linking, typing, code gen.)
- automatic storage management (GC)
- environmental queries (reflection, stack walking)
- exotic primitives (tailcall, bignums, call/cc)
- code management integrated with execution
- robust handling of incorrect inputs
- helpful runtime support libraries (REs, math, ...)
- a compiler (JIT and/or AOT) that understands it all

Problems

- VMs can do much more than C/C++,
  - > but not quite enough for emerging languages
  - > historically, the JVM was for Java only...
  - > (historically the x86 was for C and Pascal...)
- Language implementors are trying to reuse Vms
  - > Near-misses are experienced as “pain points”

Case Study: Scheme

M. Serrano, “Bigloo.NET: compiling Scheme to .NET CLR”, 2007

`http://www-sop.inria.fr/mimosa/Manuel.Serrano/publi/jot04/jot04.html`

- Uses the “natural style” for each platform (C/J/.N)
- Full continuations only in C (stack copy hack)
- Tailcall instruction in .N is too costly
- Closures poorly emulated by inner classes or delegates
- Bulky boxes for ints, pairs bloat the heap

Case Study: Python

Bolz & Rigo, “How to not write Virtual Machines for Dynamic Languages”, 2007

`http://dyla2007.unibe.ch/?download=dyla07-HowToNotWriteVMs.pdf`
`http://blogs.sun.com/jrose/entry/a_day_with_pypy`

- PyPy provides extreme flexibility to implementors
- Demands extreme flexibility from its back-end
- Fine-grained path JIT, contextually customized types
- JIT blocks connected with expandable switch and tailcall
- Could still make great use of a suitably factored VM...

So what's missing?

- Dynamic invocation

- And always, higher performance

So what's missing?

- Dynamic invocation
- Lightweight method objects
- Lightweight bytecode loading
- Continuations and stack introspection
- Tail calls and tail recursion
- Tuples and value-oriented types
- Immediate wrapper types
- Symbolic freedom (non-Java names)
- And always, higher performance

the Da Vinci Machine

a multi-language renaissance for the Java™ Virtual Machine architecture

`http://openjdk.java.net/projects/mlvm/`

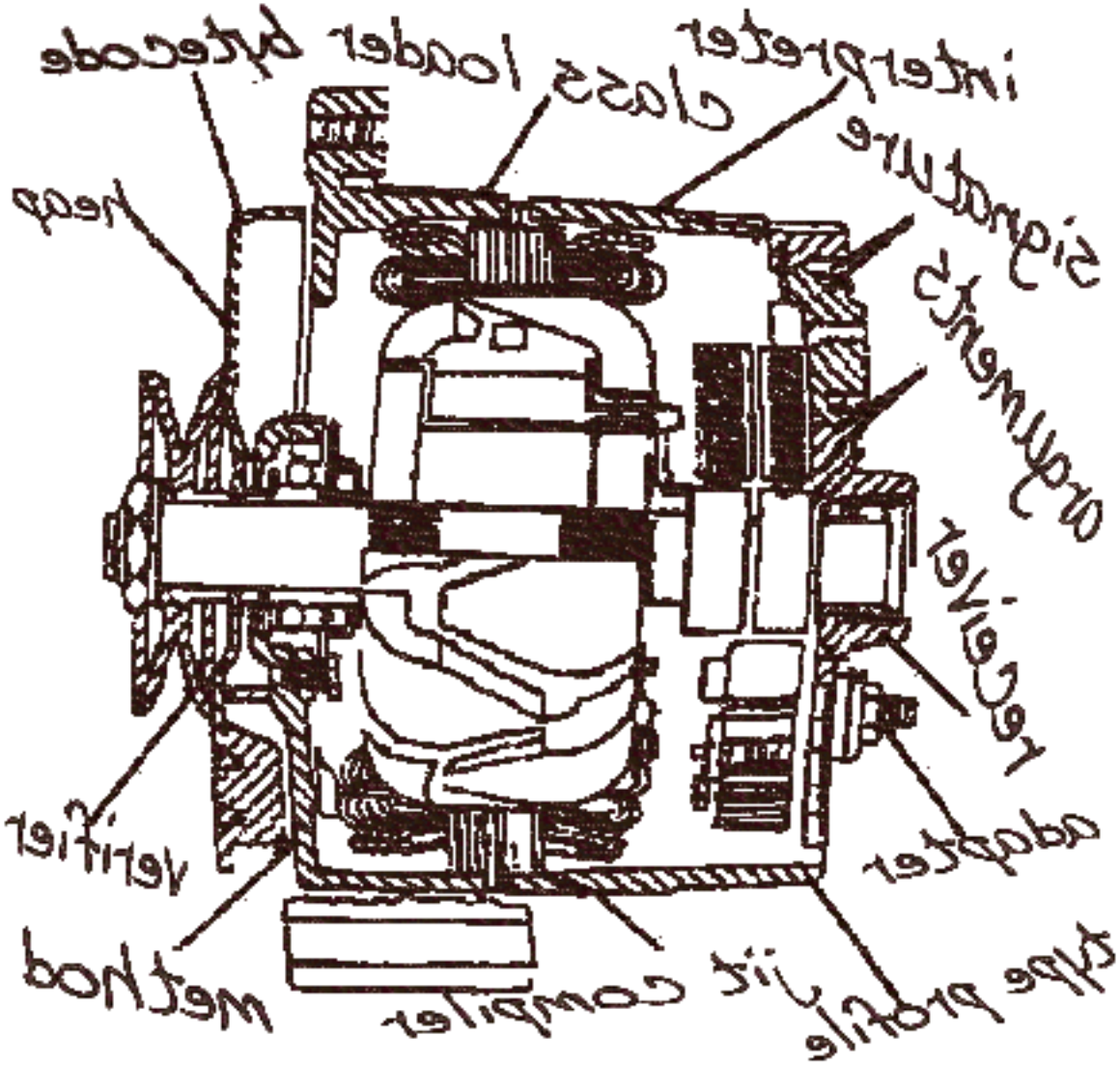
A Solution from Sun

- Evolutionary adaptation of the present JVM
- Open-ended experimentation on Sun's Hotspot
  - > wild ideas are considered, but must prove useful
  - > while incubating, features are disabled by default
- Eventual convergence on standards
- Extension of the standard JVM architecture
  - > deliberate, measured, careful extension

Da Vinci Machine Mission Statement

- Prototype JVM extensions to run non-Java languages efficiently
- First-class architectural support (not hacks or side-cars)
- Complete the existing architecture with general purpose extensions
- New languages to co-exist gracefully with Java in the JVM

Invented by Leonardo himself??



Dynamic invocation: A great idea

- non-Java call site in the bytecodes
- language-specific handler
  - > determines call linkage at runtime
  - > works in a reflective style
  - > installs direct (non-reflective) methods
- type-sensitive target method selection
- stateful: updated or revoked over time

Method handles

- Method Handle = lightweight reference to a method
- caller invokes without knowing method's name, etc.
- call runs at nearly the speed of Java call
- required to glue together dynamic call sites
- requires VM and/or library support for common adaptation patterns (curry, receiver check, varargs)

Anonymous classes

- Faster and more reliable loading and unloading
- Little interaction with system dict. or class loaders
  - > (“class names considered harmful”)
- Library-directed code customization
  - > via constant pool patching

Performance work

- No-brainer: Support less-static bytecode shapes
  - > Ongoing for years; see website for fixed bugs
  - > Examples: Class.isInstance, Arrays.copyOf
- Faster reflection
- More subtle: Faster closure-type objects
- Escape analysis (etc.) to remove auto-boxing
- Etc., etc.

Other great VM ideas
(which might need community champions)

- Interface injection (traits, mega-inheritance)
- Continuations (cf. Scheme call/cc)
- Value object (cf. Lisp fixnums)
- Tuple types (cf. .NET structs)

Are we re-inventing the world?

- No, we are adapting classic ideas to the JVM.
  - In some cases, exposing mature JVM internals to language implementors, for the first time.
  - In other cases, adjusting JVM architecture to be less Java-centric.
- Language implementors know what they want
  - (and know how to simulate it with 100x slowdown)
- VM implementors know what VMs can do
  - (and know how to make their favorite language sing)
- Let's bring them together.

Ruby meets Duke

Charles Nutter, Sr. Staff Engineer

charles.nutter@sun.com

http://headius.blogspot.com/

JRuby Design: Lexer and Parser

- Hand-written lexer
  - originally ported from MRI
  - many changes since then
- LALR parser
  - Port of MRI's YACC/Bison-based parser
    - We use Jay, a Bison for Java
  - DefaultRubyParser.y => DefaultRubyParser.java
- Abstract Syntax Tree similar to MRI's
  - we've made a few changes/additions

JRuby Design: Core Classes

- Mostly 1:1 core classes to Java types
  - String is RubyString, Array is RubyArray, etc
- Annotation-based method binding

```
public @interface JRubyMethod {
    String[] name() default {};
    int required() default 0;
    int optional() default 0;
    boolean rest() default false;
    String[] alias() default {};
    boolean meta() default false;
    boolean module() default false;
    boolean frame() default false;
    boolean scope() default false;
    boolean rite() default false;
    Visibility visibility() default
                                    Visibility.PUBLIC;
}
...
@JRubyMethod(name = "open", required = 1, frame = true)
```

JRuby Design: Interpreter

- Simple switch-based AST walker
- Recurses for nested structures
- Most code starts out interpreted
  - command-line scripts compiled immediately
  - precompiled scripts (.class) instead of .rb
  - eval'ed code always interpreted (for now)
- Reasonably straightforward code
- Future: generate the interpreter to reduce overhead

JRuby Compiler

- First complete Ruby 1.8 compiler for a general VM
- Fastest 1.8-compatible execution available
- AOT mode
  - Avoids JIT warmup time
  - Works well with “compile, run” development
  - Maybe faster startup in future? (a bit slower right now)
- JIT mode
  - Fits with typical Ruby “just run it” development
  - Eventually as fast as AOT
  - You don't have to do anything different

Compiler Pain

- AOT pain
  - Code bodies as Java methods need method handles
    - Generated as adapter methods...see JIT below
  - Ruby is very terse...i.e. compiled output is verbose
  - Mapping symbols safely (class, package, method names)
- JIT pain
  - Method body must live on a class
    - Class must be live in separate classloader to GC
    - Class name must be unique within that classloader
    - Gobs of memory used up working around all this

Compiler Optimizations

- Preallocated, cached Ruby literals
- Java opcodes for local flow-control where possible
  - Explicit local “return” as cheap as implicit
  - Explicit local “next”, “break”, etc simple jumps
- Java local variables when possible
  - Methods and leaf closures
    - leaf == no contained closures
  - No eval(), binding(), etc calls present
- Monomorphic inline method cache
  - Polymorphic for 1.1 (probably)

Optimization Pain

- “Build-your-own” dynamic invocation (always)
  - Naïve approach fails to perform (hash lookup, reflection)
- “B-y-o” reflective method handle logic
  - Handle-per-method means class+classloader per
  - Overloaded signatures means more handles
  - Non-overloading languages introduce arg boxing cost
- “B-y-o” call site optimizations
  - ...and make sure they don't interfere with JVM optz!
- We shouldn't have to worry about all this

Custom Core Classes

- String as copy-on-write byte[] impl
- Array as copy-on-write Object[] impl
- Fast-read Hash implementation
- Java “New IO” (NIO) based IO implementation
  - Example: implementing analogs for libc IO functions
- Two custom Regexp implementations
  - New one works with byte[] directly

JRuby Design: Threading

- JRuby supports only native OS threads
  - Much heavier than Ruby's green threads
  - But truly parallel, unlike Ruby 1.9 (GIL)
- Emulates unsafe green operations
  - Thread#kill, Thread#raise inherently unsafe
  - Thread#critical impossible to guarantee
  - All emulated with checkpoints (pain...)
- Pooling of OS threads minimizes spinup cost
  - Spinning up threads from pool as cheap as green
  - Future: used for coroutine support (Ruby 1.9's “Fiber”)

JRuby Design: Extensions, POSIX

- Normal Ruby native extensions not supported
  - Maybe in future, but Ruby API exposes too much
- Native libraries accessible with JNA
  - Not JNI...JNA = Java Native Access
  - Programmatically load libs, call functions
  - Similar to DL in Ruby
  - Could easily be used for porting extensions
- JNA used for POSIX functions not in Java
  - Filesystem support (symlinks, stat, chmod, chown, ...)
  - Process control

Questions?

Let's talk...

John Rose
Charles Nutter

`{john.rose,charles.nutter}@sun.com`