A Renaissance VM: One Platform, Many Languages

John R. Rose, Da Vinci Machine Project Lead
Announcing

Java 7 support for dynamic languages:

Invokedynamic

Thursday, July 7, 2011
Overview...

• Why dynamic languages?
• The invokedynamic instruction
• Method handles
• User experience
Why dynamic languages?

• Fast turnaround time for simple programs
  – no compile step required
  – direct interpretation possible
  – loose binding to the environment

• Data-driven programming
  – program shape can change along with data shape
  – radically open-ended code (plugins, aspects, closures)
Key dynamic languages on the JVM

- JavaScript (Rhino)
- Ruby (JRuby)
- Python (Jython)
- Lisp (Clojure, Kawa, ABCL, etc.)
- Groovy
- Smalltalk
- ...and many, many more
Dynamic languages are here to stay

Source: http://tiobe.com

Thursday, July 7, 2011
“Businesses like Twitter, LinkedIn, and RedHat are increasing attention to Ruby because of its fast turnaround times... Implementations like JRuby have started to solve performance problems of the past.”

Charles Nutter
JRuby Lead, Engine Yard
Overview...

- Why dynamic languages?
- The invokedynamic instruction
- Method handles
- User experience
What a JVM can do...

<table>
<thead>
<tr>
<th>Compiler Tactics</th>
<th>Language-Specific Techniques</th>
<th>Loop Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delayed Compilation</td>
<td>Class Hierarchy Analysis</td>
<td>Loop Unrolling</td>
</tr>
<tr>
<td>Tiered Compilation</td>
<td>Devirtualization</td>
<td>Loop Peeling</td>
</tr>
<tr>
<td>On-Stack Replacement</td>
<td>Symbolic Constant Propagation</td>
<td>Safe Point Elimination</td>
</tr>
<tr>
<td>Delayed Reoptimization</td>
<td>Autobox Elimination</td>
<td>Iteration Range Splitting</td>
</tr>
<tr>
<td>Program Dependence Graph Representation</td>
<td>Escape Analysis</td>
<td>Range Check Elimination</td>
</tr>
<tr>
<td>Static Single Assignment Representation</td>
<td>Lock Elision</td>
<td>Loop Vectorization</td>
</tr>
<tr>
<td>Proof-Based Techniques</td>
<td>Exact Type Inference</td>
<td>Global Code Shaping</td>
</tr>
<tr>
<td>Exact Type Inference</td>
<td>Memory Value Inference</td>
<td>Inlining (Graph Integration)</td>
</tr>
<tr>
<td>Memory Value Tracking</td>
<td>Constant Folding</td>
<td>Global Code Motion</td>
</tr>
<tr>
<td>Constant Folding</td>
<td>Reassociation</td>
<td>Heat-Based Code Layout</td>
</tr>
<tr>
<td>Operator Strength Reduction</td>
<td>Null Check Elimination</td>
<td>Switch Balancing</td>
</tr>
<tr>
<td>Null Check Elimination</td>
<td>Type Test Strength Reduction</td>
<td>Throw Inlining</td>
</tr>
<tr>
<td>Type Test Elimination</td>
<td>Algebraic Simplification</td>
<td>Control Flow Graph Transformation</td>
</tr>
<tr>
<td>Algebraic Simplification</td>
<td>Common Subexpression Elimination</td>
<td>Local Code Scheduling</td>
</tr>
<tr>
<td>Common Subexpression Elimination</td>
<td>Integer Range Typing</td>
<td>Local Code Bundling</td>
</tr>
<tr>
<td>Integer Range Typing</td>
<td>Flow-Sensitive Rewrites</td>
<td>Delay Slot Filling</td>
</tr>
<tr>
<td>Flow-Sensitive Rewrites</td>
<td>Conditional Constant Propagation</td>
<td>Graph-Coloring Register Allocation</td>
</tr>
<tr>
<td>Dominating Test Detection</td>
<td>Expression Hoisting</td>
<td>Linear Scan Register Allocation</td>
</tr>
<tr>
<td>Flow-Carried Type Narrowing</td>
<td>Expression Sinking</td>
<td>Live Range Splitting</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td>Redundant Store Elimination</td>
<td>Copy Coalescing</td>
</tr>
<tr>
<td></td>
<td>Adjacent Store Fusion</td>
<td>Constant Splitting</td>
</tr>
<tr>
<td></td>
<td>Card-Mark Elimination</td>
<td>Copy Removal</td>
</tr>
<tr>
<td></td>
<td>Merge-Point Splitting</td>
<td>Address Mode Matching</td>
</tr>
<tr>
<td></td>
<td>Merge-Point Splitting</td>
<td>Instruction Peepholing</td>
</tr>
<tr>
<td></td>
<td>Memory and Placement Transformation</td>
<td>DFA-Based Code Generator</td>
</tr>
<tr>
<td></td>
<td>Expression Hoisting</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Expression Sinking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Redundant Store Elimination</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adjacent Store Fusion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Card-Mark Elimination</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Merge-Point Splitting</td>
<td></td>
</tr>
</tbody>
</table>
...And what slows down a JVM

• Non-Java languages require special call sites.
  – Example: Smalltalk message sending (no static types).
  – Example: JavaScript or Ruby method call (different lookup rules).

• In the past, special calls required simulation overheads
  – …such as reflection and/or extra levels of lookup and indirection
  – …which have inhibited JIT optimizations.

• Result: Pain for non-Java developers.

• Enter Java 7.
Key Features

• New bytecode instruction: *invokedynamic*.
  – Linked reflectively, under user control.
  – User-visible object: `java.lang.invoke.CallSite`
  – Dynamic call sites can be linked and relinked, dynamically.

• New unit of behavior: *method handle*
  – The content of a dynamic call site is a method handle.
  – Method handles are function pointers for the JVM.

  – (Or if you like, each MH implements a single-method interface.)
Dynamic program composition

- Bytecodes are created by Java compilers or dynamic runtimes.
- The JVM seamlessly integrates execution, optimizing to native code as necessary.
- Each call site is bound to one or more method handles, which point back to bytecoded methods.
- A dynamic call site is created for each invokedynamic bytecode.

Bytecodes

Dynamic call sites

JVM

JIT

Method handles
Passing the burden to the JVM

• Non-Java languages require special call sites.
• In the past, special calls required simulation overheads

• Now, invokedynamic call sites are fully user-configurable
  – ...and are fully optimizable by the JIT.
• Result: Much simpler code for language implementors
  – ...and new leverage for the JIT.
What’s in a method call? *(before invokedynamic)*

<table>
<thead>
<tr>
<th></th>
<th>Source code</th>
<th>Bytecode</th>
<th>Linking</th>
<th>Executing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naming</td>
<td>Identifiers</td>
<td>Utf8 constants</td>
<td>JVM “dictionary”</td>
<td></td>
</tr>
<tr>
<td>Selecting</td>
<td>Scopes</td>
<td>Class names</td>
<td>Loaded classes</td>
<td>V-table lookup</td>
</tr>
<tr>
<td>Adapting</td>
<td>Argument conversion</td>
<td>C2I / I2C adapters</td>
<td>Receiver narrowing</td>
<td></td>
</tr>
<tr>
<td>Calling</td>
<td></td>
<td></td>
<td></td>
<td>Jump with arguments</td>
</tr>
</tbody>
</table>

© 2011 Oracle Corporation
### What’s in a method call? *(using invokedynamic)*

<table>
<thead>
<tr>
<th>Source code</th>
<th>Bytecode</th>
<th>Linking</th>
<th>Executing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naming</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Selecting</td>
<td>∞</td>
<td>Bootstrap methods</td>
<td>∞</td>
</tr>
<tr>
<td>Adapting</td>
<td>∞</td>
<td>Method handles</td>
<td>∞</td>
</tr>
<tr>
<td>Calling</td>
<td></td>
<td>Jump with arguments</td>
<td></td>
</tr>
</tbody>
</table>

© 2011 Oracle Corporation

Thursday, July 7, 2011
“Invokedynamic is the most important addition to Java in years. It will change the face of the platform.”

Charles Nutter
JRuby Lead, Engine Yard
Overview...

• Why dynamic languages?
• The invokedynamic instruction
• Method handles
• User experience
Invokedynamic “plumbing”, take 1

```java
aload_1; aload_2
invdyn lessThan:Z
if_icmpeq
...
```

This pointer links to the target method, a "Method Handle"

Class Runtime

```java
lessThan(, )Z:
...
```

© 2011 Oracle Corporation
More details about method handles

• A *direct method handle* points to a Java method.
  – A DMH can emulate any of the pre-existing invoke instructions.

• A *bound method handle* includes an saved argument.
  – The bound argument is specified on creation, and is used on call.
  – The bound argument is inserted into the argument list.
  – Any MH can be be bound, and the binding is invisible to callers.

• An *adapter method handle* adjusts values on the fly.
  – Both argument and return values can be adjusted.
  – Adaptations include cast, box/unbox, collect/spread, filter, etc.
  – Any MH can be adapted. Adaptation is invisible to callers.
Invokedynamic “plumbing”, take 2

```java
class Runtime {
    invoke_2(String message, Object, Object):
        ...
    
    toBoolean Adapter
    Bound MH
    String "lessThan"
    
    this chain of targets converts a return value to boolean, and inserts an extra message argument

    aload_1;aload_2
    invdyn lessThan:Z
    if_icmpeq
    ...
```
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;
...
MethodHandles.Lookup LOOKUP = lookup();

MethodHandle HASHCODE = LOOKUP
    .findStatic(System.class,
        "identityHashCode", methodType(int.class, Object.class));
{assertEquals("xy".hashCode(), (int) HASHCODE.invoke("xy"));}

MethodHandle CONCAT = LOOKUP
    .findVirtual(String.class,
        "concat", methodType(String.class, String.class));
{assertEquals("xy", (String) CONCAT.invokeExact("x", "y"));}

MethodHandle CONCAT_FU = CONCAT.bindTo("fu");
{assertEquals("futbol", CONCAT_FU.invoke("tbol"));}
Overview...

- Why dynamic languages?
- The invokedynamic instruction
- Method handles
- User experience
“We were able to implement all of the Smalltalk constructs... using invokedynamic to execute Smalltalk code on the JVM. ...The ease of putting a true dynamic language on the JVM was a wonder in itself.”

Mark Roos
Roos Instruments, Inc.
What shall we build today?

• Smaller, simpler script engines on the JVM.

• New options for high-performance programming.
  – Multiple programming paradigms with full optimization.
  – Function pointers, self-adjusting code.

• The only limit is our community’s imagination.
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Over the next hill...

• Project Lambda: mastering the multi-core
  – Direct support for closures via invokedynamic.
  – Better optimization of parallel, data-intensive programs.
And the next...

• Da Vinci Machine Project: an open source incubator for JVM futures
  – Yearly event: JVM Language Summit
  – http://openjdk.java.net/projects/mlvm/jvmlangsummit/
“Invokedynamic makes it possible for every static-typed operation on JVM to be dynamic... my mind boggles at the possibilities.”

Charles Nutter
JRuby Lead, Engine Yard