Towards a Renaissance VM

John Rose
Brian Goetz
Sun Microsystems
The JVM has, in large part, been the engine behind the success of the Java Language. In years to come, it will power the success of other languages too.
Virtual Machines

A *virtual machine* is a software implementation of a specific computer architecture

- Could be a real hardware architecture or a fictitious one

*System virtual machines* simulate a complete computer system

- VMWare, VirtualBox, VirtualPC, Parallels
- Usually implement a real hardware architecture (e.g., X86)
Virtual Machines

> Process virtual machines host a single application
  - Such as the JVM
  - Usually implement a fictitious instruction set designed for a specific purpose
  - Instruction set can be chosen to maximize implementation flexibility

> Initial implementations of VMs is often interpreted (and slow)
  - More mature implementations allow sophisticated compilation techniques
Virtual Machines

> VM isolates the application from the host system
  ● VM appears as an ordinary host process
  ● Isolate applications running in separate VMs from each other

> VM isolates the host system from the application
  ● VM acts as intermediary between hosted application and host system
  ● Hosted application can only access resources provided by the VM
Virtual Machines

> "Every problem in computer science can be solved by adding an additional layer of indirection"

- VMs are an indirection layer between application and underlying platform
- Provides portability across platforms
- Abstract from low-level architectural considerations
  - Size of register set, hardware word size
- In the 1990s, we spent a lot of time looking for the ideal ANDF (architecture-neutral distribution format)
  - Goal: Write once, run anywhere
  - Secondary goal: not language-specific
VMs have won the day

> Today, it is silly for a compiler to target hardware
  ● Much more effective to target a VM
  ● Writing a native compiler is lots more work!

> Languages need runtime support
  ● C runtime is tiny and portable (and wimpy!)
  ● More sophisticated language runtimes need
    ● Memory management, security, reflection, concurrency control, libraries, tools (debuggers, profilers, etc)
  ● In VM-based systems, many of these features are baked into the VM
VMs have won the day

> From the compiler-writer's perspective…
  
  ● If the VM doesn't provide these features, you could:
    ● Reinvent them yourself (lots of work!)
    ● Do without them
  
  ● If the VM does provide them, you'll use them
    ● Less work
    ● Makes your programming language better
  
  ● Targeting an existing VM also reuses libraries, tools
    ● Platform features such as reflection, threads
    ● APIs for tools (JVMTI), management, monitoring
    ● Rich ecosystems of tools (debuggers, profilers, IDEs)
Lots of VMs out there

- Java Virtual Machine
- CLR
- Dalvik
- Smalltalk
- Perl
- Python
- YARV
- Rubinius
- Tamarin (ActionScript)
- Valgrind (C++)
- Lua
- LLVM
- TrueType
- Parrot (Perl 6)
- Flash
- p-code (USCD Pascal)
- Squeak
- Zend (php)
VMs and performance

> First version of VMs usually interpreted, and slow
  ● But this is a "V1 implementation artifact"
> VMs favor dynamic compilation over static
  ● Delays compilation until runtime
  ● Often yields better performance
  ● Has more information available to make optimization decisions
    ● Online profiling information gathered during runtime
    ● Whole-program information
      ● Knowledge of which classes are loaded right now
    ● Knowledge of target hardware architecture
      ● Cache line size, NUMA, exotic primitives
VMs and performance

> Dynamic compilation offers techniques not available to static compilers
  - Adaptive techniques – Choose between spin-locking and blocking on a per-lock-site basis based on profiling data
  - Speculative techniques – Compile optimistically, deoptimize when proven necessary

> Powerful dynamic compilers enable static compilers to generate “dumb” code
  - The VM will optimize it better at runtime anyway!
  - Every language immediately gains the benefit of the VM's dynamic compiler
The Great Ruby Shootout, 2008

Geometric mean of the ratios (101 benchmarks)

- Ruby 1.8.6 (Vista)
- Rubinius
- JRuby 1.1.6RC1
- REE
- Ruby 1.9.1
- Ruby 1.8.7 (apt-get)
- Ruby 1.8.7 (from source)
The virtuous circle

> When multiple languages target a VM, VM facilities become common across languages
  * Java code can reflectively call JRuby code
  * Java objects and Jython objects are garbage-collected together

> All sorts of good things come out of this
  * Multi-language development
    * Use the right tool for the job, while sharing infrastructure
  * Extensibility – e.g., let users extend a Java app with Python scriptlets
  * Migration – start using Ruby without giving up existing investment
Languages on the JVM

- Groovy
- JavaFX Script
- Jython
- Scala
- C#
- Forth
- PHP
- Java
- JavaScript
- Lisp
- Basic
- Scheme
- Basic
- C
- Eiffel
- Groovy
- JESS
- Modula-2
- Nice
- Prolog
- Smalltalk
- JHCR
- Scheme
- Jickle
- Coq
- Fun
- Python
- Processing
- BeanShell
- Tiger
- LOGO
- Java

Languages on the JVM
JVM Architecture

> Stack-based representation of program
  ● Abstracts away from any specific hardware
    ● Especially register set size – moves register allocation responsibility to VM
    ● Effectively a postorder traversal of AST

> Core instructions
  ● Stack and local variable management
  ● Arithmetic, logical, conversion, comparison ops
  ● Method invocation and field access/assignment
  ● Object + array creation, exceptions, monitors

> Data types
  ● Eight primitive data types, objects, and arrays
JVM Architecture

- Object model: single inheritance with interfaces
  - Method resolution statically typed for both parameters and return value
- Dynamic linking + static type checking
  - Requires runtime verification
- Sounds a lot like the Java language
  - But...some of this is only skin-deep
  - 200 compiler writers can't be wrong!
"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."
JVM Specification, circa 1997

"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." (Removed in 2ed :-)

Java: VM vs. Language

Java language fictions

JVM features

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

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

Open classes
Dynamic typing
'eval'
Closures
Mixins
Regular expressions

Ruby language fictions

Primitive types+ops
Object model
Access control
Memory model
Dynamic linking
GC
Unicode
Compiler-based language fictions

Language fictions are easy to implement in compiler

- Some are costless
  - Checked exceptions in Java
  - Name lookup in Java – *done at link time*
  - Method selection (overloads!) – *done at compile time*

- Some are easy but difficult to implement *efficiently*
  - Open classes in Ruby
  - Duck typing in Python
  - Numeric towers in Scheme
Compiler-based language fictions

> It is easy to work around VM limitations
  > ...if you don't care about performance
  >   Dynamic calls via reflection
  >   Multiple dispatch via Visitor pattern or unrolled switch
  >   Build numeric types atop BigInteger

> Great for getting to an implementation quickly
  > Users are initially pretty happy, then the hate mail starts
Pain points for dynamic languages

> What makes life hard for languages on the JVM?
  * Method invocation (including linkage and selection)
  * New viewpoints on Java types (List versus LIST)
  * Boxed values (Integer versus fixnum)
  * Special control structures (tailcall, generators, etc.)
  * (…and so on, but the series converges quickly…)

> Why are these things hard?
  * The JVM has its generic features
  * Languages have their special fictions
  * Differences are simulated (complexity, overhead)
Pain points for dynamic languages (2)

- Removing the workarounds ➔ a maze of puzzles
  - No direct way to eliminate simulation overheads
  - New, more clever simulations are more complex
  - Months of experimenting with alternatives
- The VM offers a limited set of generic features
  - Features set has been initially stretched to fit Java
  - Design is tasteful, generic, but incomplete
  - Good news: Adding a little more gains us a lot!
  - Each additional “stretch” fits many more languages
Pain points for dynamic languages

> If we could make one change to the JVM to make life better for dynamic languages, what would it be?

> Answer: flexible (non-static) method calls!

> (Runner up: Interface injection. To discuss later…)
The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)
The deal with method calls (in one slide)

- Calling a method is cheap (VMs can even inline!)
- Selecting the right target method can be costly
  - Static languages do most of their method selection at compile time (e.g., `System.out.println(x)`)
    - Single-dispatch on receiver type is left for runtime
  - Dynamic langs do almost none at compile-time
    - But it would be nice to not have to re-do method selection for every single invocation
The deal with method calls (in one slide)

> Calling a method is cheap (VMs can even inline!)
> Selecting the right target method can be costly
> Static languages do most of their method selection at compile time (e.g., `System.out.println(x)`)  
  > Single-dispatch on receiver type is left for runtime
> Dynamic langs do almost none at compile-time  
  > But it would be nice to not have to re-do method selection for every single invocation
> Each language has its own ideas about linkage
> The VM enforces static rules of naming and linkage  
  > Language runtimes want to decide (& re-decide) linkage
What’s in a method call?

> Naming — using a symbolic name
> Linking — reaching out somewhere else
> Selecting — deciding which one to call
> Adapting — agreeing on calling conventions

> (...and finally, a parameterized control transfer)
A connection from caller A to target B

- Including naming, linking, selecting, adapting:
- ...where B might be known to A only by a name
- ...and A and B might be far apart
- ...and B might depend on arguments passed by A
- ...and a correct call to B might require adaptations

(After everything is decided, A jumps to B’s code.)

a) names are subjected to Java scoping & access
Example: Fully static invocation

For this source code

```java
String s = System.getProperty("java.home");
```

The compiled byte code looks like

```
0: ldc #2           //String "java.home"
2: invokestatic #3  //Method java/lang/System.getProperty:
(Ljava/lang/String;)Ljava/lang/String;
5: astore_1
```
Example: Fully static invocation

> For this source code

```java
String s = System.getProperty("java.home");
```

The compiled byte code looks like

```
0: ldc #2           //String "java.home"
2: invokestatic #3  //Method java/lang/System.getProperty:
               (Ljava/lang/String;)Ljava/lang/String;
5: astore_1
```

a) Names are embedded in the bytecode
b) Linking handled by the JVM with fixed Java rules
c) Target method selection is not dynamic at all
d) No adaptation: Signatures must match exactly
How the VM sees it:

```java
ldc "java.home"
invstat Sys.getProperty(S)
```

(Note: This implementation is typical; VMs vary.)
How the VM sees it:

(Note: This implementation is typical; VMs vary.)
How the VM sees it:

(Note: This implementation is typical; VMs vary.)
Example: Class-based single dispatch

> For this source code

```java
//PrintStream out = System.out;
out.println("Hello World");
```

> The compiled byte code looks like

```
4:  aload 1
5:  ldc #2            //String "Hello World"
7:  invokevirtual #4  //Method java/io/PrintStream.println:
    (Ljava/lang/String;)V
```
Example: Class-based single dispatch

> For this source code

```java
//PrintStream out = System.out;
out.println("Hello World");
```

> The compiled byte code looks like

```
4:   aload_1
5:   ldc #2            //String "Hello World"
7:   invokevirtual #4  //Method java/io/PrintStream.println:
             (Ljava/lang/String;)V
```

a) Again, names in bytecode
b) Again, linking fixed by JVM
c) *Only* the receiver type determines method selection
d) *Only* the receiver type can be adapted (narrowed)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
How the VM selects the target method:

(Note: This implementation is typical; VMs vary.)
What more could anybody want? (1)

> Naming — not just Java names
  > arbitrary strings, even structured tokens (XML??)
  > help from the VM resolving names is optional
  > caller and callee do not need to agree on names
What more could anybody want? (1)

- Naming — not just Java names
  - arbitrary strings, even structured tokens (XML??)
  - help from the VM resolving names is optional
  - caller and callee do not need to agree on names

- Linking — not just Java & VM rules
  - can link a call site to any callee the runtime wants
  - can re-link a call site if something changes
What more could anybody want? (1)

> Naming — not just Java names
  > arbitrary strings, even structured tokens (XML??)
  > help from the VM resolving names is optional
  > caller and callee do not need to agree on names

> Linking — not just Java & VM rules
  > can link a call site to any callee the runtime wants
  > can re-link a call site if something changes

> Selecting — not just static or receiver-based
  > selection logic can look at any/all arguments
  > (or any other conditions relevant to the language)
What more could anybody want? (2)

> Adapting — no exact signature matching
>   widen to Object, box from primitives
>   checkcast to specific types, unbox to primitives
>   collecting/spreading to/from varargs
>   inserting or deleting extra control arguments
>   language-specific coercions & transformations
What more could anybody want? (2)

- Adapting — no exact signature matching
  - widen to Object, box from primitives
  - checkcast to specific types, unbox to primitives
  - collecting/spreading to/from varargs
  - inserting or deleting extra control arguments
  - language-specific coercions & transformations

- (…and finally, the same fast control transfer)
What more could anybody want? (2)

- Adapting — no exact signature matching
  - widen to Object, box from primitives
  - checkcast to specific types, unbox to primitives
  - collecting/spreading to/from varargs
  - inserting or deleting extra control arguments
  - language-specific coercions & transformations

- (…and finally, the same fast control transfer)
- (…with inlining in the optimizing compiler, please)
(Doesn’t Java already give us all this?)

- Yes, but only statically
  - (like scoping, overloading, autoboxing)
  - Many languages want to do similar things on the fly.
- Java compilers perform selection & adaptation.
  - When the JVM sees the bytecodes for Java, all that static work is done (early binding).
  - Other languages want late binding and rebinding.
(But, don’t interfaces fill the gap?)

- Interfaces are indeed more powerful than classes
- You get a few more degrees of freedom in linkage
  - ...due to interface “multiple inheritance”

- But as with classes you get:
  - only single dispatch, receiver-based selection
  - no argument adaptation (except receiver narrowing)
  - several indirections (via “itables” instead “vtables”)
Dynamic method invocation

How would we compile a function like

```javascript
function max(x, y) {
    if (x.lessThan(y)) then y else x
}
```
Dynamic method invocation

How would we compile a function like

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

Specifically, how do we call `.lessThan()`?
Dynamic method invocation (how not to)

> How about:

0:   aload_1; aload_2
2:   invokevirtual #3  //Method Unknown.lessThan:
      (LUnknown;)Z
5:   if_icmpeq

> That doesn't work

> No receiver type
> No argument type
> Return type might not even be boolean (‘Z’)
Dynamic method invocation (slowly)

How about:

0:   aload_1; aload_2
2:   ldc #2  // "lessThan"
4:   invokestatic #3  //Method my/Runtime.invoke_2:
     (Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
5:   invokestatic #4  //Method my/Runtime.toBoolean:
     (Ljava/lang/Object;)Z
8:   if_icmpeq

That works, but … really … slowly

Argument types are (re-)computed remotely
Less possibility of local caching or optimization
Lots of indirections!
(“Invoker” objects? Similar problems + complexity.)
Dynamic method invocation (how to)

> A new option:

```
0:  aload_1; aload_2
2:  invokedynamic  #3  //NameAndType lessThan:
    (Ljava/lang/Object;Ljava/lang/Object;)Z
>  5:  if_icmpeq
```

> Advantages:

- Compact representation
- Argument types are untyped Objects
Dynamic method invocation (how to)

> A new option:

```
0:  aload_1; aload_2
2:  invokedynamic #3  //NameAndType lessThan:
    (Ljava/lang/Object;Ljava/lang/Object;)Z
>  5:  if_icmpeq
```

> Advantages:

- Compact representation
- Argument types are untyped Objects
- Required boolean return type is respected
- (Flexibility from *signature polymorphism.*)
Dynamic method invocation (details)

> But where is the dynamic language plumbing??
> We need something like `invoke_2` and `toBoolean`!
> How does the runtime know the name `lessThan`?
Dynamic method invocation (details)

But where is the dynamic language plumbing??

> We need something like `invoke_2` and `toBoolean`!
> How does the runtime know the name `lessThan`?

Answer, part 1: it’s all *method handles* (MH).

> A MH can point to any accessible method
> The target of an `invokedynamic` is a MH
invokedynamic, as seen by the VM:

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

**Fine print:**
- **pink thing** = CallSite object
- **green thing** = MethodHandle object
- **this is the simplest case (no adaptation)**
invokedynamic, as seen by the VM:

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

- this pointer links to the target method...
- ...and this target can change over time

class Runtime

```java
lessThan(,)Z: 
... 
```
invokedynamic, as seen by the VM:

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

this pointer links to the target method...
the VM JIT compiler may venture to inline it, hoping it won't change

...and this target can change over time

```
class Runtime
lessThan(,)Z:
...
```
invokedynamic, as seen by the VM:
What’s this method handle thing?

> Fundamental unit of “pluggable” behavior
  > Directly supports linkage to any method.
  > (Can also do normal receiver-based dispatch)
  > Works with every VM signature type.
    > (Not just (Object...) => Object )
  > Can be composed in chains (like pipelines).
Dynamic method invocation (details 2)

> But can I adjust names, types, arguments?
> Answer, part 2: In short, **yes**.
> Method handles can be constructed, combined, and invoked in all kinds of useful patterns.
>   > Direct MH points to one method (maybe virtual)
>   > Adapter MH can adjust argument types on the fly
>   > Bound MH can insert extra arguments on the fly
>   > “Java MH” can run arbitrary code during a call
>   > MHs can be chained (and the chains can be inlined)
more invokedynamic plumbing:

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

toBoolean Adapter

Bound MH

String "lessThan"

class Runtime

direct MH

invoke_2(String message, Object, Object):
...

more invokedynamic plumbing:

```java
al0ad_1; al0ad_2
invdyn lessThan:Z
if_icmpeq
...
```

- `aload_1; al0ad_2` are load instructions.
- `invdyn lessThan:Z if_icmpeq` represents an invocation using dynamic plumbing.

**Diagram Explanation:**

- The triangle represents a node in the compiled byte code graph.
- `toBoolean Adapter` converts the output to a boolean value.
- `Bound MH` binds the method handle.
- `String "lessThan"` provides a string value.
- The `class Runtime` contains the `invoke_2` method which takes a `String` message and two `Object` parameters.

- The `direct MH` indicates a direct method handle lookup.

- The diagram explains that this chain of targets converts a return value to boolean, and inserts an extra message argument.

- The code snippet shows a method `invoke_2` which is called internally with a string message and two object parameters.
Dynamic method invocation (details 3)

- But who sets up these nests of method handles?
  - We need a hook for the runtime to set them up
  - And it needs to be a localizable, modular hook
Dynamic method invocation (details 3)

> But who sets up these nests of method handles?
> We need a hook for the runtime to set them up
> And it needs to be a localizable, modular hook
> Answer, part 3: a bootstrap method (BSM).
> Classes containing `invokedynamic` declare BSMs
> BSM is called the first time a given instruction runs
> The BSM gets to see all the details (name & type)
> The BSM is required to construct call-site plumbing
Dynamic method invocation (details 3)

- But who sets up these nests of method handles?
  - We need a hook for the runtime to set them up
  - And it needs to be a localizable, modular hook
- Answer, part 3: a *bootstrap method* (BSM).
  - Classes containing `invokedynamic` declare BSMs
  - BSM is called the first time a given instruction runs
  - The BSM gets to see all the details (name & type)
  - The BSM is required to construct call-site plumbing
- (Yes, it too is a method handle.)
invokedynamic bootstrap logic:

... 
aload_1; aloa_2
invdyn lessThan:Z
if_icmpeq
...

the invokedynamic instruction has not yet been executed

class Runtime

bootstrap(info...):
...
return new CallSite(info)
invokedynamic bootstrap logic:

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

the containing class must declare a bootstrap method to initialize its call sites on demand

```java
class Runtime

  bootstrap(info...):
  ...
  return new CallSite(info)
```

the invokedynamic instruction has not yet been executed
### A Budget of Invokes

<table>
<thead>
<tr>
<th>invoke-</th>
<th>invoke-</th>
<th>invoke-</th>
<th>invoke-</th>
<th>invoke-</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>special</td>
<td>virtual</td>
<td>interface</td>
<td>dynamic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>receiver</th>
<th>class</th>
<th>interface</th>
<th>no receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>no receiver</td>
<td>receiver class</td>
<td>receiver interface</td>
<td>no receiver</td>
</tr>
<tr>
<td>no dispatch</td>
<td>no dispatch</td>
<td>single dispatch</td>
<td>single dispatch</td>
</tr>
<tr>
<td>adapter-based dispatch</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| B8 nn nn | B7 nn nn | B6 nn nn | B9 nn nn aa 00 | BA nn nn 00 00 |
Method handle details

> Like methods, can have any function type
> An object of static type `java.dyn.MethodHandle`
> Unlike (other) objects, signature-polymorphic
> Like methods, can be virtual, static, or “special”
> Unlike methods, not named
> Invoked like methods (via `invokevirtual`):
  `((MethodHandle)mh).invoke(args)`
> Or (reflective style via `invokestatic`):
  `MethodHandles.invoke(mh, args)`
Speed, speed, and speed

- We are doing this to make languages run faster
  - more compact code and runtimes
  - simpler, more direct “plumbing” of calls
  - *direct application of existing VM optimizations!*
- The Java VMs have *centuries* of programmer effort invested in making the bytecodes zoom.
  - (And zoom scalably and reliably, too.)
- Why should Java have all the fun?
Optimizing single-dispatch (1)

- The VM can do a lot to speed up method dispatch
- Early binding is the über-trick
  - Early binding usually allows the option of inlining
  - …which, in turn, sets the optimizer free!
    - caller & callee are merged; optimizer gets a wider “horizon”
- Key Question: What do we choose to know early?
- Example: Devirtualization
  - Prove that the target links to a unique method
  - Optimize away entire method dispatch (and inline!)
Optimizing single-dispatch (2)

- Monomorphic inline caching
  - Observe that the call selects a unique target
  - Cache (receiver type, direct target) in code
  - Receiver type = cached type ➔ run target directly
  - Our early binding might fail; what then?
    - ➔ do something slow (but still correct) – relink to vtable
  - Can generalize this, e.g., to multiple receiver types
- This is typical VM behavior: fast path/slow path
  - speculate a common case and optimize (fast path)
  - defend against the uncommon case (slow path)
Optimizing single-dispatch (3)

- Profile-driven inline caching
  - Observe that one target is *highly probable*
  - Do the receiver check trick for that target
  - On the slow path, rinse and repeat
  - After 1-2 highly probable targets are exhausted, use a slower “megamorphic” call (vtable based)

- In a VM, fast and slow paths can shift over time
  - a slow path might cause *deoptimization*
  - …followed by *reoptimization* of the affected calls

*(If you didn’t catch all that, see session TS-5427, “Inside Out VM”.*
Optimizing invokedynamic

- Most of the old optimization framework applies!
  - (Except class hierarchy analysis.)
- Inline caching: Nail down the target method
  - Every `invokedynamic` site links to one target
  - Virtual-direct targets get the good old treatment
Optimizing invokedynamic (2)

> Inlining (the big early-binding payoff)
  > Call sites contain chains of method handles
  > …which can be inlined by the optimizer
  > …thus inlining language-specific call logic

> ➔ First class optimization of language specifics like lazy linking, partial argument binding, dispatch, coercion, etc.
Optimizing invokedynamic (3)

- Type profiling: Compiler can use call-site profiles
  - Virtual-direct MHs can profit, just as before
  - Language-specific guard logic can also profit
  - Adapters applying casts can be optimized

- Deoptimization: More code patterns can benefit
  - The JIT can optimize guard adapters that never fail
  - Arbitrary fast/slow patterns can \texttt{relink} on failure

$\Rightarrow$ the VM can now assist with \textit{speculative} optimizations that are \textit{language-specific}
Episodic runtime support

- The language runtime can assist in method resolution when needed
  - And get out of the way when no longer needed!

- Runtime can adjust binding decisions on the fly
  - early-bound (direct-to-Java) targets
  - partially-bound targets (via partial evaluation)
  - late-bound (caching or profiling) targets
  - asynchronously reset targets
And now a word from our standardizers

> (It’s not all just VM hacking...)
JSR 292 – the “invokedynamic” JSR

- Original charter of JSR-292 was to provide a specific bytecode for dynamic method invocation, plus (optionally) mechanisms for code evolution
  - The scope has widened since that
  - Current work includes
    - `invokedynamic` bytecode
      - Application-defined linkage (and re-linkage)
    - Method handles
      - All-purpose lightweight construct for "code pointers"
    - Interface injection (likely)
      - Add new methods and types to existing classes
    - More later? Tail calls, Continuations, ..
Interface injection (the 2nd pain point)

- Dynamically typed programs can be self-modifying
  - Class or method definition can change at runtime
    - Self-modifying code is hard to optimize, and dangerous
  - So, don't restructure classes, just relabel them
    - Enter Interface injection
    - Limited ability to modify old classes
      - Just enough for them to implement new interfaces
    - interface supertypes are cheap for JVM objects
    - `invokeinterface` is fast these days
a statically defined interface:
an injected interface:

class String
...

impl MyJDynMeta
  meta(): ...

MH


class MyJDynRuntime
  stringMeta(): ...

... ldc "aString" invint MyJDyn.meta() istore 5 ...

Punching through to the virtual metal

- A few judiciously chosen new features
  - let language implementations avoid simulations
    - (simpler, faster runtime support)
  - give languages more equal footing on VM with Java
  - enable new uses for old JVM optimizations