Invokedynamic in Practice

Adding invokedynamic support to JRuby

Intro

- Charles Nutter
  - @headius
  - headius@headius.com
  - JRuby guy at Engine Yard
  - JVM enthusiast
- JRuby
  - Ruby language on JVM
  - Pushes JVM/platform in many ways

JRuby Challenges

- Dynamic method dispatch
- Rich set of literals
- Dynamic "constant" lookup
  - Mutable "constants"
- Heavy use of closures
  - Heap-based structures to support closures
- Cross-call state
  - Caller can modify parts of callee frame

JRuby Challenges

- **Dynamic method dispatch**
- **Rich set of literals**
- **Dynamic "constant" lookup**
    - Mutable "constants"
- Heavy use of closures
    - Heap-based structures to support closures
- Cross-call state
    - Caller can modify parts of callee frame

tl;dr

- Invokedynamic works beautifully
  - Eliminate lots of generated code
  - Eliminate hand-written call path tweaking (arity, etc)
  - Eliminate inlining-killing pass-throughs
    - Which you hand-wrote to avoid generating!
  - Reduce bytecode in pipeline
    - And deep discounts against inlining budgets
  - Arbitrarily many, arbitrarily complex call paths
  - So much more than dynamic *or* dispatch
    - Maybe least-interesting aspect now?
- Most important addition to JVM thusfar

Method Dispatch

Several types of dispatch

- Normal: def foo(a, b, c) ... x.foo(a, b, c)
- Varargs: def foo(a, *b) ... x.foo(*c)
- Attr get/set: x.value; x.value = a
- Element get/set: x[a]; x[a] = b
- Operator assignment: x.value += a; x[b] ||= c
- Super: super(a,b,c); super
- By name: x.send :foo, a
- Implicit === for case/when and begin/rescue
- Implicit type conversions

Dispatch Paths

- Ruby to "native"
  - Heaviest hit by far in typical apps
  - Most core classes are Java
- "Native" to Ruby
  - Type conversions, "hash", etc
  - Underlines need for more Ruby in core
- Ruby to Ruby
  - Heavy in libraries/frameworks that do a lot by hand
  - Mixed mode
- Ruby to Java (Java integration)
  - Often overloaded methods
  - Argument/return often converted or (un)wrapped
- Java to Ruby (Embedding)
  - someObject.callMethod("name", arg1, arg2)

Method Binding

- Method table is fully dynamic
  - Classes start out empty
  - Methods can be added, removed, aliased any time
- Types of bound methods
  - "Native" implemented in Java code
  - Ruby methods, interpreted and jitted
  - Java methods from Java integration
- DynamicMethod
  - Superclass of all bound methods
  - Arity-split up to three arguments
  - Usually **generated bytecode** to aid inlining
    - Usually **class-per-method**
  - 2740 pre-generated in jruby.jar
    - 21MB uncompressed, 1.5MB compressed!

Current non-indy dispatch -Xcompile.invokedynamic=false

- Monomorphic inline cache
  1. All classes have a serial number
  2. Mutation of a class cascades serial update
  3. Call site caches single [serial, method] tuple
  4. Guard confirms class serial matches tuple
- Pro
  - Simple
  - Eliminates hash hit
  - Works on all JVMs
- Con
  - Cache logic defeats inlining
  - Hand-written per-arity call paths
  - Hand-written specialized cache types

Monomorphic cache example

```
Ruby: a = 1; foo(a)
...
   ALOAD 0
   INVOKEVIRTUAL ruby/__dash_e__.getCallSite0 ()
Lorg/jruby/runtime/CallSite;
   ALOAD 1
   ALOAD 2
   ALOAD 2
   ALOAD 9
   INVOKEVIRTUAL org/jruby/runtime/CallSite.call
(Lorg/jruby/runtime/ThreadContext;
Lorg/jruby/runtime/builtin/IRubyObject;
Lorg/jruby/runtime/builtin/IRubyObject;
Lorg/jruby/runtime/builtin/IRubyObject;)
Lorg/jruby/runtime/builtin/IRubyObject;
```

Monomorphic cache example

Ruby: a = 1; foo(a)

```
...
    public IRubyObject call(ThreadContext context, IRubyObject
caller, IRubyObject self, IRubyObject arg1) {
        RubyClass selfType = pollAndGetClass(context, self);
        CacheEntry myCache = cache;
        if (CacheEntry.typeOk(myCache, selfType)) {
            return myCache.method.call(context, self, selfType,
methodName, arg1);
        }
        return cacheAndCall(caller, selfType, context, self, arg1);
    }
```

Monomorphic cache example

Ruby: a = 1; foo(a)

```
...
  public class org.jruby.RubyFixnum$INVOKER$i$1$0$op_plus
{
  ...
  public org.jruby.runtime.builtin.IRubyObject call(...)
    Code:
      0: aload_2
      1: checkcast     #13                 // class
org/jruby/RubyFixnum
      4: aload_1
      5: aload         5
      7: invokevirtual #17                 // Method
org/jruby/RubyFixnum.op_plus:(...)
     10: areturn
```

Dynamic Optimization ("dynopt") -Xcompile.dynopt=true

- Guarded direct call
  - Interpreter's last target at JIT time
  - Guard against serial number
  - invokevirtual/static directly in jitted code
  - Fallback path is monomorphic cache
- Pro
  - Greatly improved dispatch performance
  - Inlining across dynopt'ed calls
  - Fallback no worse than monomorphic cache
- Con
  - Significantly more code per call (2x+)
  - Consumes too much inlining budget
  - Double-guard for non-monomorphic calls
  - Can't work across classloaders

Dynopt example

Ruby: def foo; foo; end; foo

```
...
    ALOAD 2
    LDC 579
    INVOKESTATIC org/jruby/javasupport/util/RuntimeHelpers.
isGenerationEqual(...)
    IFEQ L2
    ALOAD 0
    ALOAD 1
    ALOAD 2
    ACONST_NULL
    INVOKESTATIC
rubyjit/foo_7F1E71544C0BFF52B6020F56F3C0D1A11E173AF5.__file__ (...)
    GOTO L3
   L2
    ALOAD 0
    INVOKEVIRTUAL
rubyjit/foo_7F1E71544C0BFF52B6020F56F3C0D1A11E173AF5.getCallSite0
```

Invokedynamic Dispatch
-Xcompile.invokedynamic=true

- Best of all worlds
  - Guard using class serial again
  - Monomorphic calls use MethodHandle only
  - Polymorphic calls form a PIC
  - Megamorphic calls degrade to inline cache
- Pro
  - Greatly reduced bytecode
  - No opto-busting intermediate code
  - Inlining across dyncalls
  - No need for generated DynamicMethods (usually)
- Con
  - Only Java 7 (or via backport)
  - Not fully optimized yet

Method Dispatch

Several types of dispatch

- **Normal: def foo(a, b, c) ... x.foo(a, b, c)**
- Varargs: def foo(a, *b) ... x.foo(*c)
- Attr get/set: x.value; x.value = a
- **Element get/set: x[a]; x[a] = b**
- Operator assignment: x.value += a; x[b] ||= c
- Super: super(a,b,c); super
- By name: x.send :foo, a
- **Implicit === for case/when and begin/rescue**
- Implicit type conversions

Dispatch Paths

- **Ruby to "native"**
  - Heaviest hit by far in typical apps
  - Most core classes are Java
- "Native" to Ruby
  - Type conversions, "hash", etc
  - Underlines need for more Ruby in core
- **Ruby to Ruby**
  - Heavy in libraries/frameworks that do a lot by hand
  - Mixed mode
- **Ruby to Java (Java integration)**
  - Often overloaded methods
  - Argument/return often converted or (un)wrapped
- Java to Ruby (Embedding)
  - someObject.callMethod("name", arg1, arg2)

Invokedynamic example

Ruby: def foo(a, b); a + b; end

```
...
    ALOAD 1
    ALOAD 2
    ALOAD 10
    LDC "+"
    ALOAD 11
    INVOKEDYNAMIC call (...)[invocationBootstrap(...)]
    ARETURN
```

Invokedynamic example

Ruby: def foo(a, b); a + b; end

...

```
   public static CallSite invocationBootstrap(Lookup lookup, String name, MethodType type)
throws NoSuchMethodException, IllegalAccessException {
       CallSite site = new JRubyCallSite(lookup, type, CallType.NORMAL, false, false, true);

       MethodType fallbackType = type.insertParameterTypes(0, JRubyCallSite.class);
       MethodHandle myFallback = insertArguments(
               lookup.findStatic(InvokeDynamicSupport.class, "invocationFallback",
               fallbackType),
               0,
               site);
       site.setTarget(myFallback);
       return site;
   }
```

Invokedynamic example

Ruby: def foo(a, b); a + b; end

...

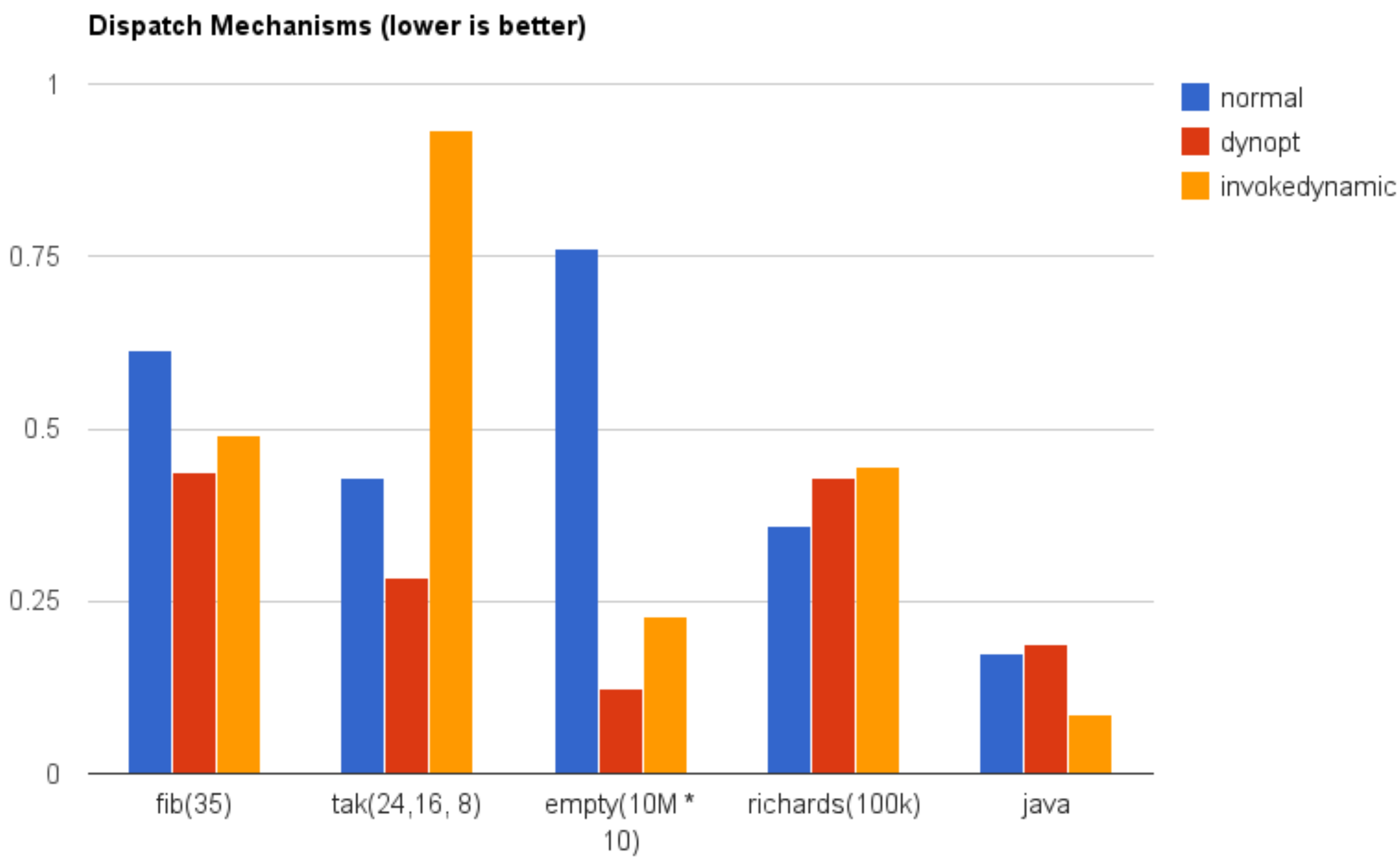
```
   public static IRubyObject invocationFallback(JRubyCallSite site, ThreadContext context,
IRubyObject caller, IRubyObject self, String name, IRubyObject arg0) throws Throwable {
       RubyClass selfClass = pollAndGetClass(context, self);
       CacheEntry entry = selfClass.searchWithCache(name);
       // method_missing logic elided
       MethodHandle target = getTarget(site, selfClass, name, entry, 1);

       // bind target into site...next slide

       return (IRubyObject)target.invokeWithArguments(context, caller, self, name, arg0);
   }
```

Invokedynamic example

Ruby: def foo(a, b); a + b; end

...

```
    if (target == null || ++site.failCount > RubyInstanceConfig.MAX_FAIL_COUNT)
{
        site.setTarget(target = createFail(FAIL_1, site, name, entry.method));
    } else {
        target = postProcess(site, target);
        if (site.getTarget() != null) {
            site.setTarget(createGWT(TEST_1, target, site.getTarget(), entry, site,
false));
        } else {
            site.setTarget(createGWT(TEST_1, target, FALLBACK_1, entry, site));
        }
    }
```



Dispatch Mechanisms (lower is better)
1
0.75
0.5
0.25
0
fib(35)
tak(24,16, 8)
empty(10M * 10)
richards(100k)
java
normal
dynopt
invokedynamic

Literals, etc

- Numbers
  - Fixnum
  - Float
  - Bignum
- Character data
  - Strings
  - Symbols
  - Regexp
- Collections
  - Array
  - Hash
- Closures
- Call sites

Current JRuby

- Cache object associated with method
  - Each method associated with RuntimeCache instance
  - RuntimeCache holds arrays of literals
  - Literals allocated a script load or lazily
- Pro
  - Simple
  - Avoids initializing unused literals
  - Reasonably fast
- Con
  - Many levels of indirection
  - 2-3 field or array dereferences

Literals example

Ruby: 1

...

```
    ALOAD 0
    ALOAD 1
    BIPUSH 100
    INVOKEVIRTUAL ruby/__dash_e__.getFixnum0
(Lorg/jruby/runtime/ThreadContext;I)Lorg/jruby/RubyFixnum;
    ARETURN
```

Literals example

Ruby: 1

...

```
public class class AbstractScript extends Script {
    ...
    public final RubyFixnum getFixnum(ThreadContext context, int i, int value) {
        return runtimeCache.getFixnum(context, i, value);
    }
    ...
    public final RubyFixnum getFixnum0(ThreadContext context, int value) {
        return runtimeCache.getFixnum(context, 0, value);
    }
```

Literals example

Ruby: 1

...

```
  public final RubyFixnum getFixnum(ThreadContext context, int index, int
value) {
      RubyFixnum fixnum = fixnums[index];
      if (fixnum == null) {
         return fixnums[index] = RubyFixnum.newFixnum(context.runtime,
value);
      }
      return fixnum;
  }
```

Invokedynamic for literals

- Literal loads using invokedynamic
  - Bootstrap as ConstantCallSite when possible
  - MutableCallSite lazily bound otherwise
- Pro
  - Value bound directly at call site
  - Reduced indirection
  - Greatly reduced bytecode
  - Better inlining characteristics
- Con
  - Not always faster right now

Invokedynamic literals

Ruby: 100

...

```
    ALOAD 1
    INVOKEDYNAMIC getFixnum (...) [InvokeDynamicSupport.getFixnumBootstrap
(...) (6), 100]
```

Invokedynamic literals

Ruby: 100

...

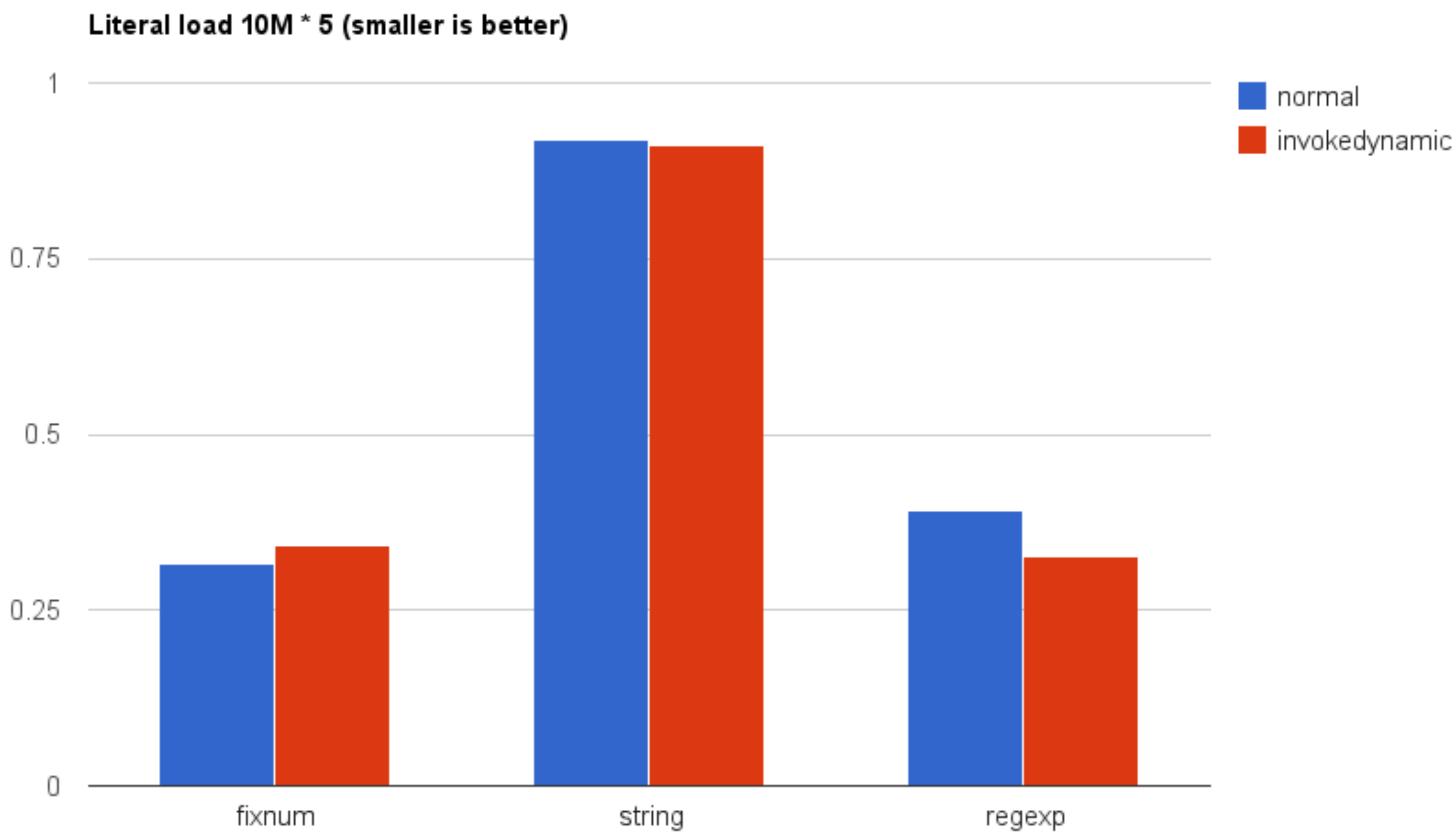
```
public static CallSite getFixnumBootstrap(Lookup lookup,
        String name, MethodType type, long value) {
    MutableCallSite site = new MutableCallSite(type);
    MethodHandle init = findStatic(
        InvokeDynamicSupport.class,
        "initFixnum",
        methodType(RubyFixnum.class,
            MutableCallSite.class,
            ThreadContext.class, long.class));
    init = insertArguments(init, 2, value);
    init = insertArguments(
        init,
        0,
        site);
    site.setTarget(init);
    return site;
}
```

Invokedynamic literals

Ruby: 100

...

```
public static RubyFixnum initFixnum(MutableCallSite site,
        ThreadContext context, long value) {
    RubyFixnum rubyFixnum =
            context.runtime.newFixnum(value);
    site.setTarget(
            dropArguments(
                constant(RubyFixnum.class,
                        rubyFixnum),
            0,
            ThreadContext.class));
    return rubyFixnum;
}
```



Literal load 10M * 5 (smaller is better)
1
0.75
0.5
0.25
0
fixnum
string
regexp
normal
invokedynamic

Constants

- Constants are set at runtime
  - Usually on script load
  - Rarely set or reset by app code
- Hierarchical and lexical
  - Lexically enclosing namespaces first
  - Current object's class hierarchy second
- Lexical nature forces global guard
  - Update a constant, global dirty bit gets flipped
  - Runtime constant update *strongly* discouraged

Current JRuby

- Local cache, global invalidate
    - Constant access caches [serial, value] tuple
    - Cached in RuntimeCache like literals
    - Serial invalidated globally
        - Constant write
        - Module mix-in
- Pro
    - Simple
    - Reasonably fast
- Con
    - Indirection and array access, like literals
    - Caching may obscure repeat accesses
    - Not inlining-friendly

Constant lookup example

Ruby code: Object

```
...
   ALOAD 0
   ALOAD 1
   LDC "Object"
   INVOKEVIRTUAL ruby/__dash_e__.getConstant0
(Lorg/jruby/runtime/ThreadContext;Ljava/lang/String;)
Lorg/jruby/runtime/builtin/IRubyObject;
   ARETURN
```

Constant lookup example

Ruby code: Object

...

```
public class RuntimeCache {
   public final IRubyObject getConstant(ThreadContext context,
          String name, int index) {
       IRubyObject value = getValue(context, name, index);

       return value != null ? value : context.getCurrentScope().getStaticScope().getModule().
callMethod(context, "const_missing", context.getRuntime().fastNewSymbol(name));
   }

   public IRubyObject getValue(ThreadContext context, String name, int index) {
       IRubyObject value = constants[index];
       return isCached(context, value, index) ? value : reCache(context, name, index);
   }
```

Constant lookup example

Ruby code: Object

```
...
    private boolean isCached(ThreadContext context,
IRubyObject value, int index) {
        return value != null && constantGenerations[index] ==
context.getRuntime().getConstantInvalidator().getData();
    }

    public IRubyObject reCache(ThreadContext context, String
name, int index) {
        Object newGeneration = context.getRuntime().
getConstantInvalidator().getData();
        IRubyObject value = context.getConstant(name);
        constants[index] = value;
        if (value != null) {
```

Invokedynamic-based constants

- SwitchPoint to the rescue!
  - Constant cache uses SwitchPoint.GWT
  - Global invalidation invalidates SwitchPoint
  - **No active guard required**
- Pro
  - No active guard
  - No userland indirection or array deref
  - Value is embedded into call site
  - Better inlining characteristics
- Con
  - SwitchPoint is still slow

Invokedynamic constant example

Ruby code: Object

...

ALOAD 1

INVOKEDYNAMIC Object (...)[InvokeDynamicSupport.getConstantBootstrap(...) (6)]

Invokedynamic constant example

Ruby code: Object

```
...
   public static CallSite getConstantBootstrap(Lookup lookup, String name, MethodType
type) throws NoSuchMethodException, IllegalAccessException {
      RubyConstantCallSite site;

      site = new RubyConstantCallSite(type, name);

      MethodType fallbackType = type.insertParameterTypes(0, RubyConstantCallSite.
class);
      MethodHandle myFallback = insertArguments(
            lookup.findStatic(InvokeDynamicSupport.class, "constantFallback",
            fallbackType),
            0,
            site);
      site.setTarget(myFallback);
      return site;
   }
```

Invokedynamic constant example

Ruby code: Object

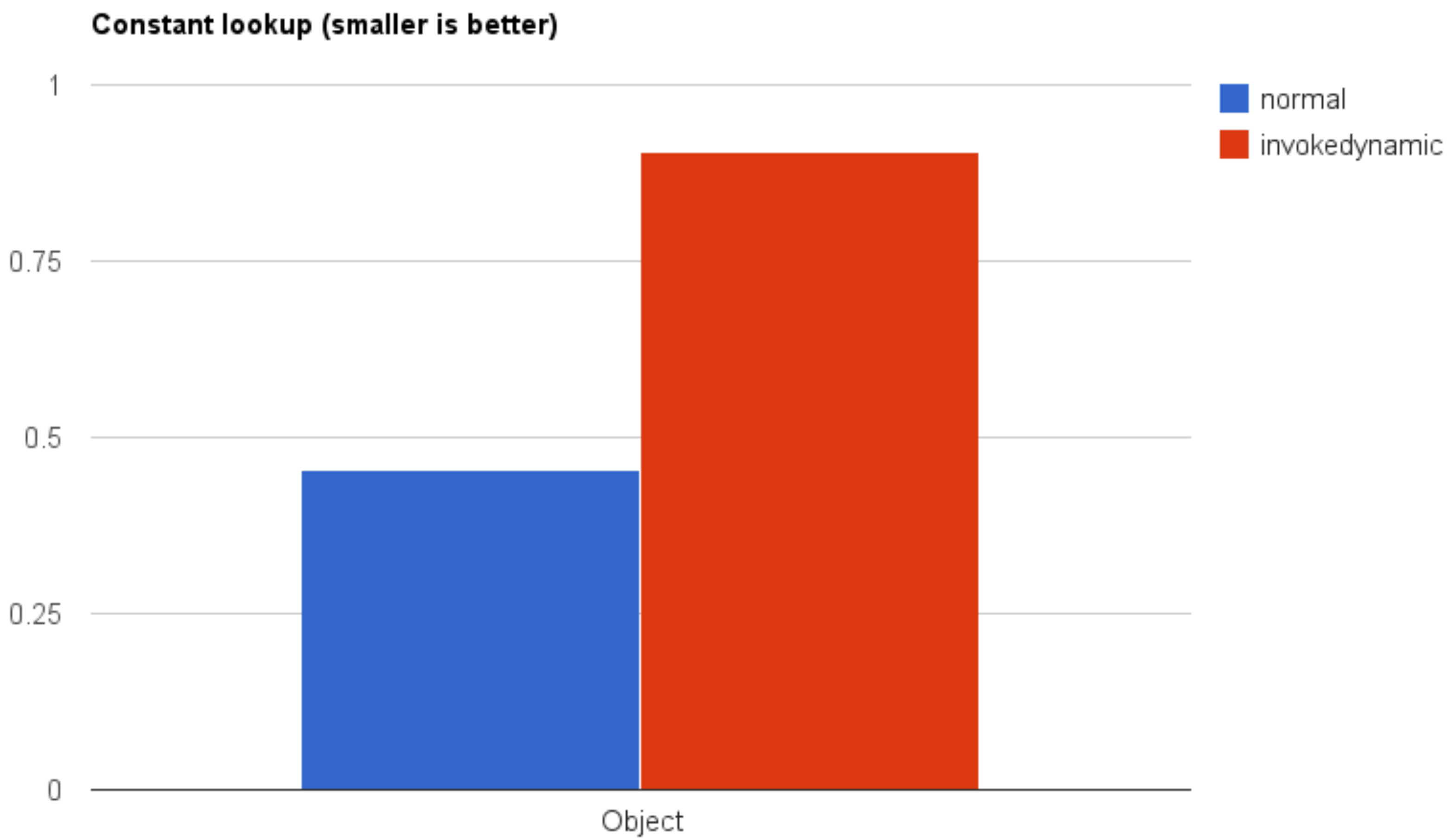
...

```
   public static IRubyObject constantFallback(RubyConstantCallSite site,
         ThreadContext context) {
      IRubyObject value = context.getConstant(site.name());

      if (value != null) {
         if (RubyInstanceConfig.LOG_INDY_CONSTANTS) LOG.info("constant "
+ site.name() + " bound directly");

         MethodHandle valueHandle =
            constant(IRubyObject.class, value);
         valueHandle =
            dropArguments(valueHandle, 0,
                  ThreadContext.class);

         MethodHandle fallback = insertArguments(
               findStatic(InvokeDynamicSupport.class, "constantFallback",
```

Invokedynamic constant example

Ruby code: Object

```
...
        MethodHandle fallback = insertArguments(
            findStatic(InvokeDynamicSupport.class,
              "constantFallback",
              methodType(IRubyObject.class,
                  RubyConstantCallSite.class,
                  ThreadContext.class)),
            0,
            site);

        SwitchPoint switchPoint = (SwitchPoint)context.runtime.
getConstantInvalidator().getData();
        MethodHandle gwt =
          switchPoint.guardWithTest(
            valueHandle, fallback);
        site.setTarget(gwt);
```


Constant lookup (smaller is better)
1
0.75
0.5
0.25
0
Object
normal
invokedynamic

Future: JRuby

- Eliminate active guard
  - Type check + SwitchPoint for modification
  - Faster?
- Remaining dispatch paths
  - All Ruby-to-X paths should be doable
  - Native to Ruby requires bytecode rewrite (at least)
  - Java to Ruby will require Java 7
    - Return MethodHandle vs x.callMethod("blah")
- Continue working with JVM folks
  - Dynopt as "ideal" perf (when it works)
  - Invokedynamic to equal invokevirtual
  - Fast as Java = more JRuby in Ruby

Future: Other

- Dynalink
  - JRuby will use it
  - Java 8 needs something like it
- Java 7-specific build of JRuby
  - Rewritten native-to-Ruby calls
  - No generated DynamicMethods
    - Smaller dist!
- Optimization promises from 292.next?
  - Or at least explicit complexity guarantees

Future: 292

- Continue optimizing
    - Better unoptimized perf
        - -client is 10x slower than server in 1.7.0
    - I will do what I can to help ;-)
        - I don't mind reading assembly
        - I want to help the other JVMs too
        - I want a continual 292 impl deathmatch
- Default implementations
    - Cookbook cases (stay tuned!)
    - Dynalink framework (stay tuned!)
        - At least JLS method selection, please!
- Inspectability
    - Debugging bad MH chains is painful
    - Writing tutorials is painful :-)

Questions?

Appendix

Why Not All Paths?

- Haven't gotten to it yet
- Framing/scoping logic adds several layers of handles
  - Not as fast as generated code as of 1mo ago
- Additional logic to arity-match
  - And generate errors on mismatch

Arity-splitting

- Generated handles split arities up to 3

```
public class RubyArray extends RubyObject {
    ...
    @JRubyMethod(name = {"[]", "slice"}, compat = RUBY1_8)
    public IRubyObject aref(IRubyObject arg0) {...}

    @JRubyMethod(name = {"[]", "slice"}, compat = RUBY1_8)
    public IRubyObject aref(
            IRubyObject arg0,
            IRubyObject arg0) {...}
```

Arity-splitting

- Generated handles split arities up to 3

```
public class RubyArray$INVOKER$i$aref extends
        JavaMethod$JavaMethodOneOrTwo {
  ...
  public IRubyObject call(ThreadContext, IRubyObject,
      RubyModule, String, IRubyObject, IRubyObject);
  public IRubyObject call(ThreadContext, IRubyObject,
      RubyModule, String, IRubyObject);
}
```

Arity-splitting

- Generated handles split arities up to 3

```
  public IRubyObject call(ThreadContext, IRubyObject, RubyModule, String,
IRubyObject, org.jruby.runtime.builtin.IRubyObject);
    Code:
      0: aload_2
      1: checkcast     #13   // class org/jruby/RubyArray
      4: aload         5
      6: aload         6
      8: invokevirtual #17   // Method RubyArray.aref:(...)
     11: areturn
```

Arity-splitting

- Generated handles split arities up to 3

```
public IRubyObject call(ThreadContext, IRubyObject,
    RubyModule, String, IRubyObject, IRubyObject);
  Code:
     0: aload_2
     1: checkcast     #13   // class org/jruby/RubyArray
     4: aload         5
     6: aload         6
     8: invokevirtual #17   // Method RubyArray.aref:(...)
    11: areturn
```

Framing

- Artificial frames on heap
  - Usually not allocated; preallocated per-thread
  - But populating on the way in/out isn't free
- Frame and "scope" separate
  - Allow omitting one or the other when possible
- Generated DynamicMethods do framing
  - So 292 handle-based dispatch must do framing
- Rare on core methods, "not uncommon" on Ruby methods
  - Ruby methods call core methods that manipulate frame
  - Core methods (almost) never get artificial frame (in 1.6)

Framing

```
@JRubyMethod(name = "raise", optional = 3, frame = true,
        module = true, visibility = Visibility.PRIVATE,
        omit = true)
 public static IRubyObject rbRaise(
        ThreadContext context, IRubyObject recv,
        IRubyObject[] args, Block block) {
```

```
  public IRubyObject call(ThreadContext, IRubyObject, RubyModule, String, IRubyObject[], Block);
    Code:
       0: aload         5
       2: arraylength
       3: ldc           #12                 // int 3
       5: if_icmpgt     11
       8: goto          25
      11: aload_1
      12: invokevirtual #18                 // Method org/jruby/runtime/ThreadContext.getRuntime:()Lorg/jruby/Ruby;
      15: aload         5
      17: ldc           #19                 // int 0
      19: ldc           #12                 // int 3
      21: invokestatic  #25                 // Method org/jruby/runtime/Arity.checkArgumentCount:(Lorg/jruby/Ruby;
[Lorg/jruby/runtime/builtin/IRubyObject;II)I
      24: pop
      25: aload_0
      26: aload_1
      27: aload_2
      28: aload         4
      30: aload         6
      32: invokevirtual #29                 // Method org/jruby/internal/runtime/methods/JavaMethod$JavaMethodNBlock.
preFrameOnly:(Lorg/jruby/runtime/ThreadContext;Lorg/jruby/runtime/builtin/IRubyObject;Ljava/lang/String;
Lorg/jruby/runtime/Block;)V
      35: aload_1
      36: aload_2
      37: aload         5
      39: aload         6
      41: invokestatic  #35                 // Method org/jruby/java/addons/KernelJavaAddons.rbRaise:
(Lorg/jruby/runtime/ThreadContext;Lorg/jruby/runtime/builtin/IRubyObject;[Lorg/jruby/runtime/builtin/IRubyObject;
Lorg/jruby/runtime/Block;)Lorg/jruby/runtime/builtin/IRubyObject;
      44: aload_1
      45: invokestatic  #39                 // Method org/jruby/internal/runtime/methods/JavaMethod$JavaMethodNBlock.
postFrameOnly:(Lorg/jruby/runtime/ThreadContext;)V
      48: areturn
      49: aload_1
      50: invokestatic  #39                 // Method org/jruby/internal/runtime/methods/JavaMethod$JavaMethodNBlock.
postFrameOnly:(Lorg/jruby/runtime/ThreadContext;)V
      53: athrow
    Exception table:
```