Towards native performance with dynamic languages on the JVM

Marcus Lagergren, Oracle

Towards native performance with dynamic languages on the JVM

How far does **invokedynamic** take us?

Marcus Lagergren, Oracle

Towards native performance with dynamic languages on the JVM

How far does **invokedynamic** take us?

Pretty far actually!

Marcus Lagergren, Oracle

The Legal Slide

"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 DECISION. THE DEVELOPMENT, RELEASE, AND TIMING OF ANY FEATURES OR FUNCTIONALITY DESCRIBED FOR ORACLE'S PRODUCTS REMAINS AT THE SOLE DISCRETION OF ORACLE."

Who am I?

@lagergren

Agenda

- Nashorn summary
- The last nine months of Nashorn and `invokedynamic` performance work
- How did we do it?
- The future

Nashorn

- Part of JDK 8
  - 100% ECMA compliant JavaScript runtime written in Java
- ECMA 5.1 support
  - Limited ECMA 6 support (8u40)
- `invokedynamic` based
- Generates bytecode
  - Currently no AST interpretation

Nashorn – release schedule

- Java 8
  - First version for public consumption

Nashorn – release schedule

- Java 8
  - First version for public consumption
- Java 8u20
  - Security fixes and several JIT / JDK improvements

Nashorn – release schedule

- Java 8
  - First version for public consumption
- Java 8u20
  - Security fixes and several JIT / JDK improvements
- Java 8u40
  - Performance release

Nashorn – release schedule

- Java 8
  - First version for public consumption
- Java 8u20
  - Security fixes and several JIT / JDK improvements
- Java 8u40
  - Performance release
- Java 9

Nashorn – release schedule

- Java 8
  - First version for public consumption
- Java 8u20
  - Security fixes and several JIT / JDK improvements
- Java 8u40
  - Performance release
- Java 9

Evangelist Blurb

“Nashorn is a really fast 100% JS compilant invokedynamic based JavaScript runtime written in Java. It runs on top of the JVM”

… or maybe?

"Nashorn is a runtime for dynamic languages on the JVM. Nashorn takes away the pain and performance loss of bridging the gap between language and bytecode. Types, objects and code are automatically adapted to be an optimal fit in the JVM mould"

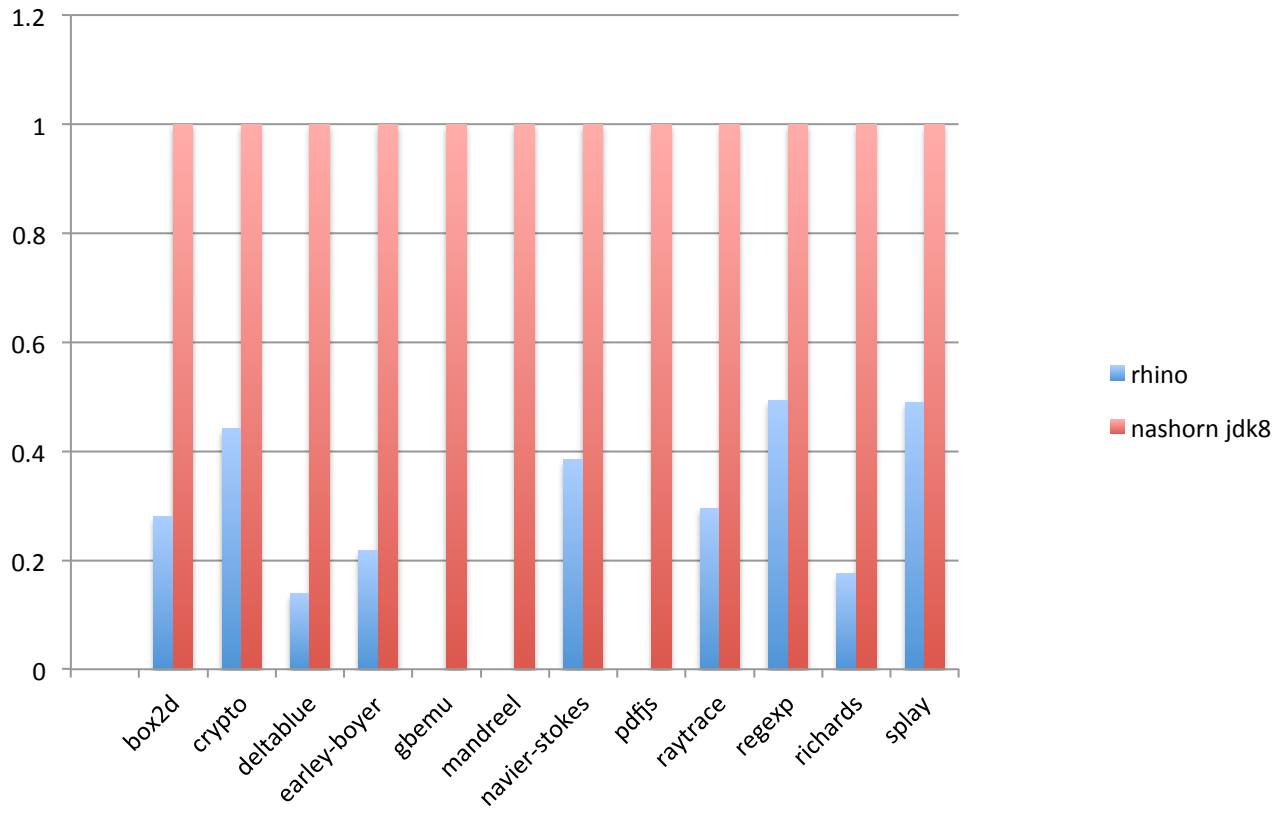
JavaScript

JavaScript

"Once you admit that EVERYTHING IS TERRIBLE, the universe opens infinitely many doors for you... ***TO TERRIBLE****."*

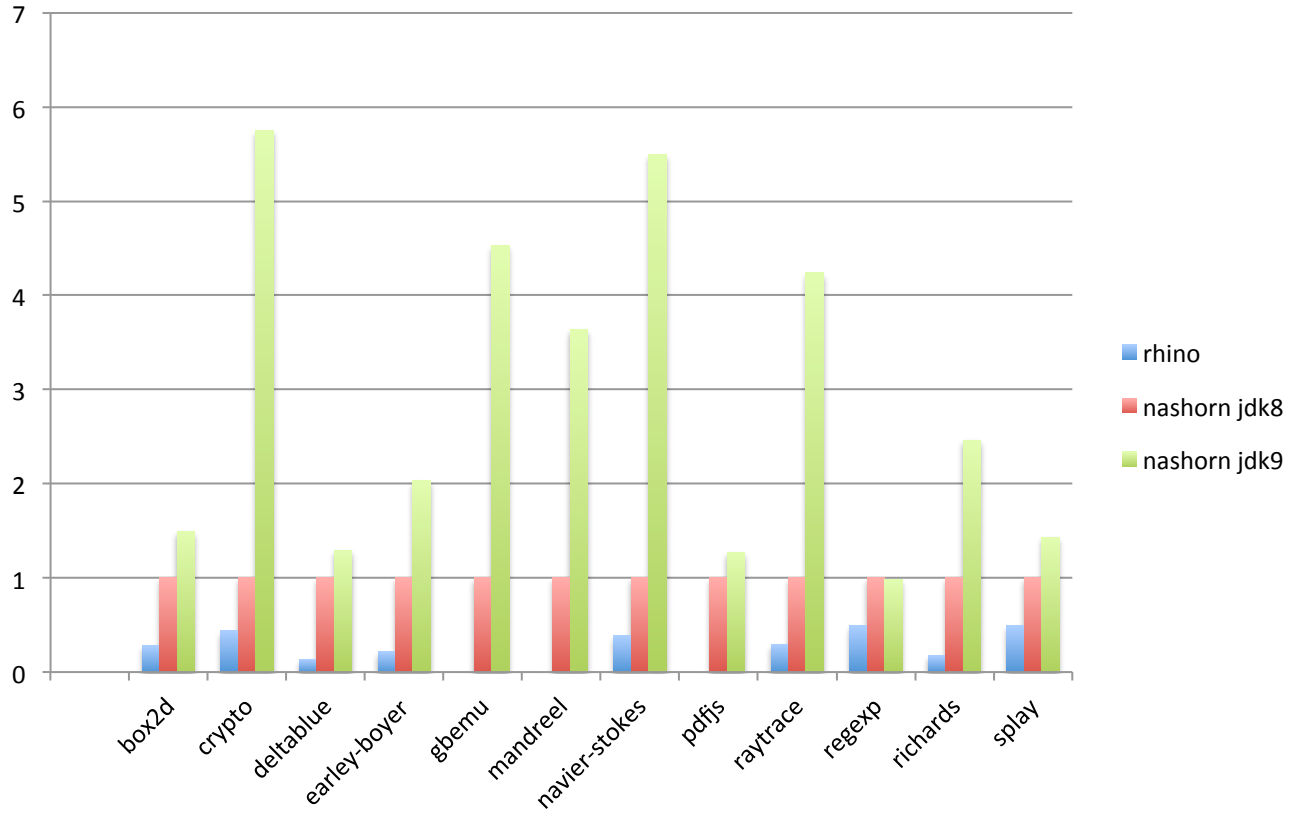
- Jerry Kuch

Where were we last October?



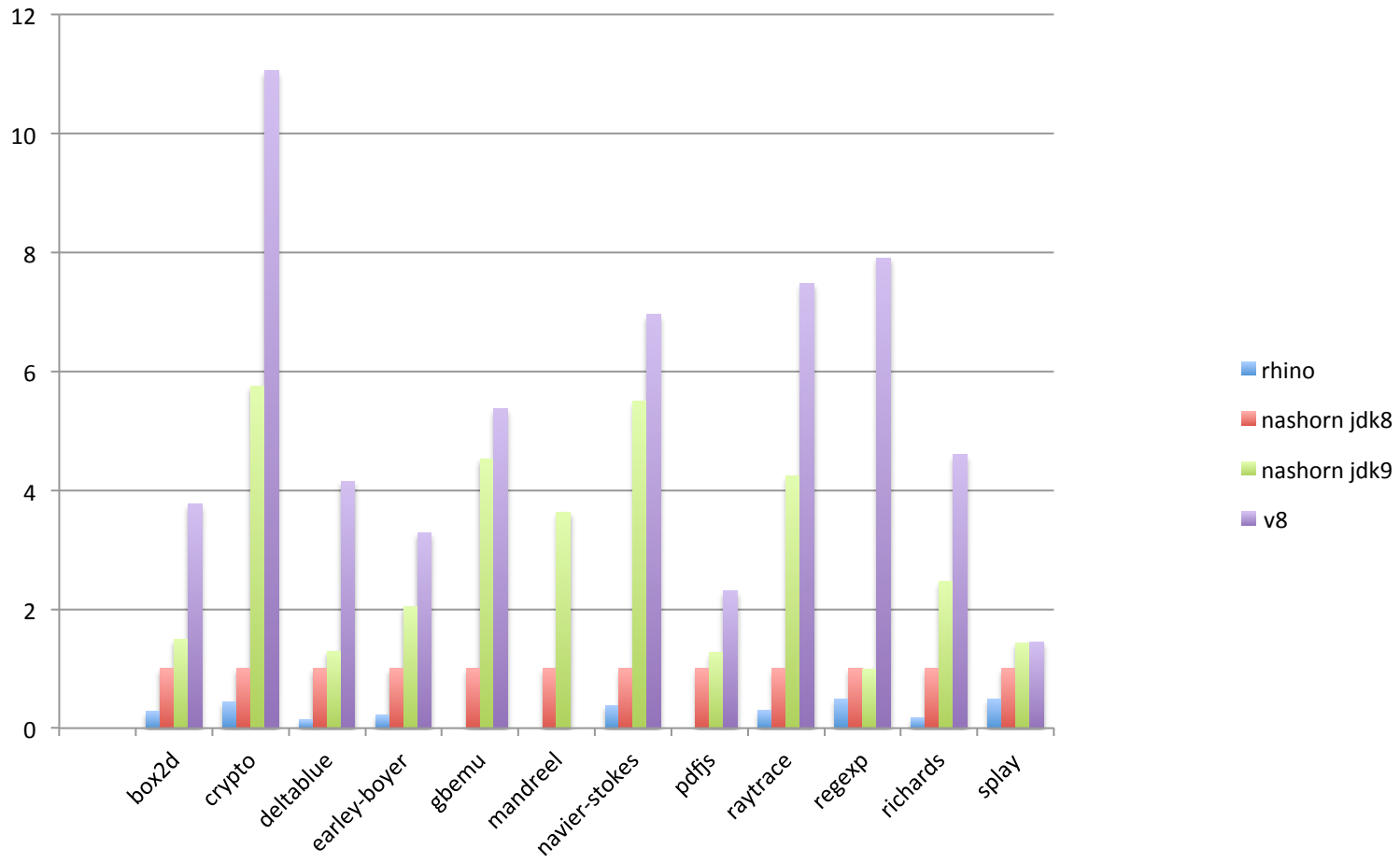
Octane Benchmark r31

Where are we today?



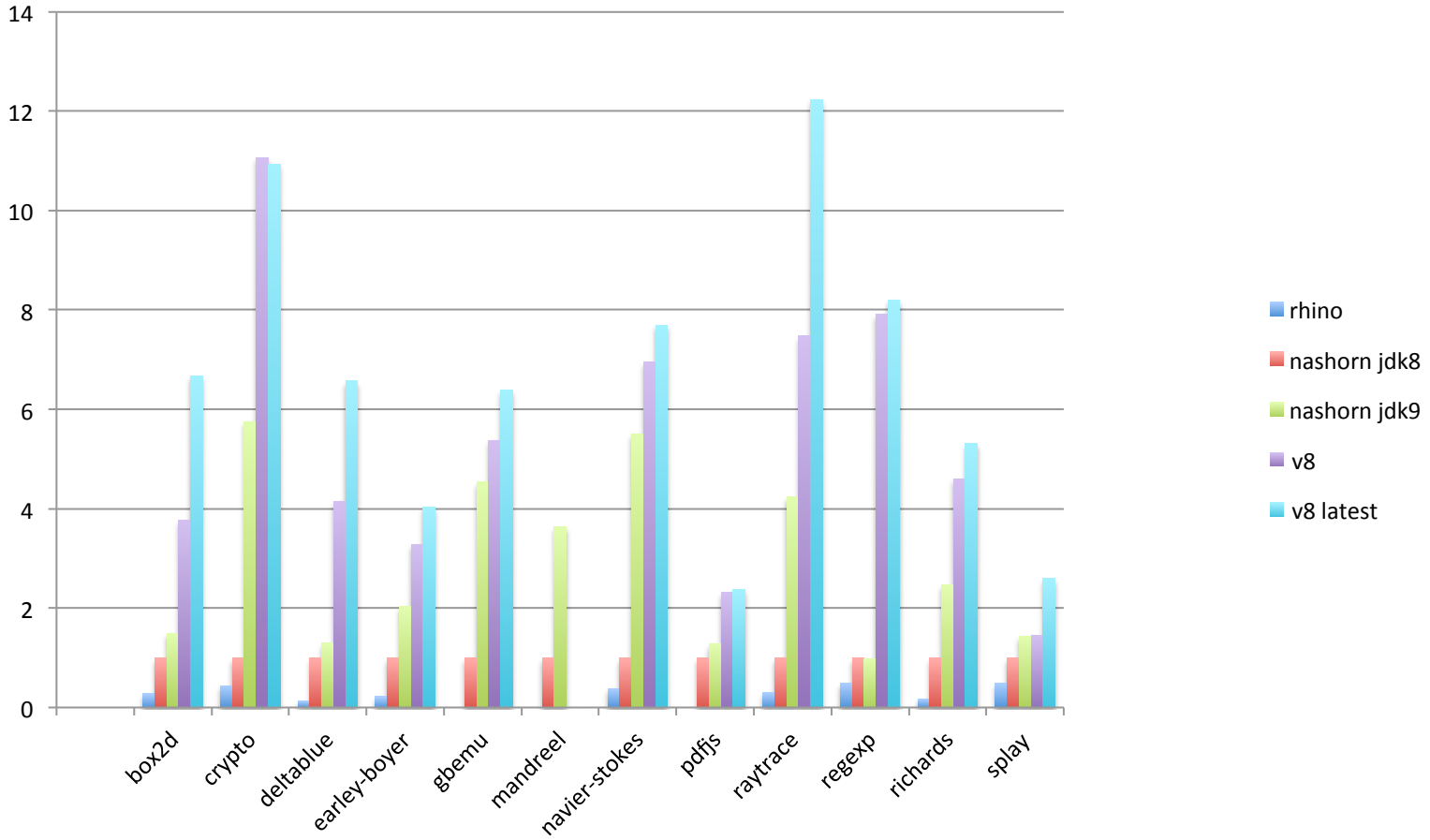
Octane Benchmark r31

… against v8



Octane Benchmark r31

v8 bleeding edge…



Octane Benchmark r31

Let’s not forget node

avatar.js

- Project Avatar
  - HTTP benchmark
  - REST benchmark
- Comparable to v8 / native performance
- 8u40 will deliver plenty of improvements
- Use of exposed mechanisms
  - `BufferArrayData`

avatar.js

Node RPS	Lyra RPS
2693.50	3103.47
6709.69	7894.39
14277.12	16209.81
26369.21	26750.27
27736.08	26716.41
27729.44	27038.73

- Project Avatar
  - HTTP benchmark
  - REST benchmark
- Comparable to v8 / native performance
- 8u40 will deliver plenty of improvements
- Use of exposed mechanisms
  - `BufferArrayData`

The last year

“The Secret Sauce”

Optimistic Type Runtime Architecture

[http://openjdk.java.net/jeps/196]

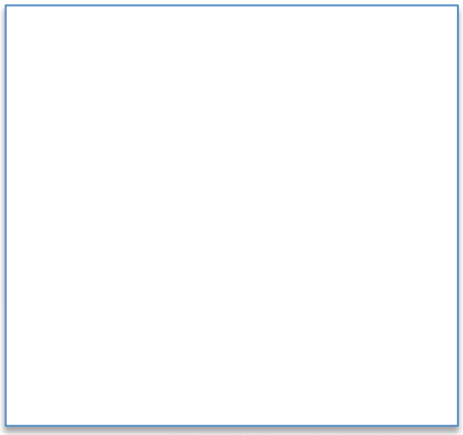
Just compile to bytecode!

- “The JVM is technically advanced and has man decades of JIT optimizations”
- “The JVM magically JITs arbitrary bytecode to fast native code”

Just compile to bytecode!

- “The JVM is technically advanced and has man decades of JIT optimizations”
- “The JVM magically JITs arbitrary bytecode to fast native code”
- Nope :-(

That moment of disappointment



when you can't find the answer
to your homework on Google

GCD

```
function gcd(x, y) {
    var a = x;
    var b = y;
    while (a != b) {
       if (a > b) {
          a = a - b;
       } else {
          b = b - a;
       }
    }
    return a;
}

gcd(x, y);
```

GCD – Naïve & JS Compliant

```
public static gcd(Object;Object;Object;)Object;

  aload 1
  astore 4
  aload 2
  astore 5
  goto L2
  aload 4
  aload 5
  invokedynamic GT:ZOO_Z(Object;Object;)Z;
  ifeq L4
  aload 4
  invokestatic JSType.toNumber(Object;)D
  aload 5
  invokestatic JSType.toNumber(Object;)D
  dsub
  invokestatic java/lang/Double.valueOf(D)Ljava/lang/Double;
  astore 4
  goto L2
  aload 5
  invokestatic JSType.toNumber(Object;)D
  aload 4
  invokestatic JSType.toNumber(Object;)D
  dsub
  invokestatic java/lang/Double.valueOf(D)Ljava/lang/Double;
  astore 5
  aload 4
  aload 5
  invokedynamic NE:ZOO_Z(Object;Object;)Z
  ifne L3
  aload 4
  areturn
```

GCD – vs Ideal

```
public static gcd(Object;Object;Object;)Object;

  aload 1
  astore 4
  aload 2
  astore 5
  goto L2
  aload 4
  aload 5
  invokedynamic GT:ZOO_Z(Object;Object;)Z;
  ifeq L4
  aload 4
  invokestatic JSType.toNumber(Object;)D
  aload 5
  invokestatic JSType.toNumber(Object;)D
  dsub
  invokestatic java/lang/Double.valueOf(D)Ljava/lang/Double;
  astore 4
  goto L2
  aload 5
  invokestatic JSType.toNumber(Object;)D
  aload 4
  invokestatic JSType.toNumber(Object;)D
  dsub
  invokestatic java/lang/Double.valueOf(D)Ljava/lang/Double;
  astore 5
  aload 4
  aload 5
  invokedynamic NE:ZOO_Z(Object;Object;)Z
  ifne L3
  aload 4
  areturn
```

```
public static gcd(Object;II)I

  iload 1
  istore 3
  iload 2
  istore 4
  iload 3
  iload 4
  if_icmpeq L10
  iload 3
  iload 4
  if_icmple L12
  iload 3
  iload 4
  isub
  istore 3
  goto L14
  iload 4
  iload 3
  isub
  istore 4
  goto L9
  iload 3
  ireturn
```

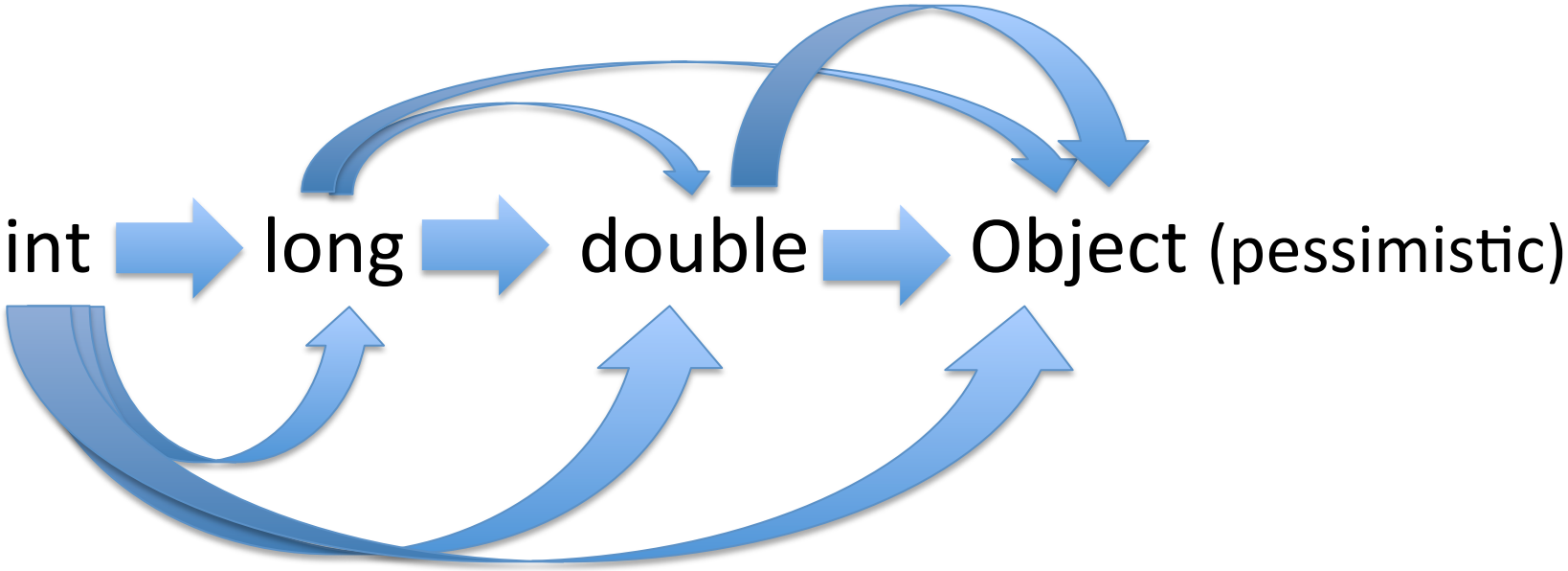
Optimistic Types

- Factor 150x difference in execution speed between the two
- “Generate Java-like bytecode”

Optimistic Types

- Generate as optimistic a function as possible
- Surround optimistic operations with exception handlers
- When a type is too narrow
  - Store frame state (no bytecode stack)
  - Take a continuation and finish method
  - Tell linker to regenerate more conservatively

Optimistic Types



Implementation

```
function f() {
  return x * y; // x,y in scope
}
```

```
public static f(ScriptFunction;Object;)D
   aload 0
   invokevirtual ScriptFunction.getScope()LScriptObject;
   astore 2  // store scope in slot 2
   aload 2   // load scope
   invokedynamic get:x(Object;)Object;
   aload 2   // load scope
   invokedynamic get:y(Object;)Object;
   swap
   invokestatic JSType.toNumber(Object;)D
   dup2_x1
   pop2
   invokestatic JSType.toNumber(Object;)D
   dmul
   dreturn
```

```
public static f(ScriptFunction;Object;)I
     aload 0
     invokevirtual ScriptFunction.getScope()ScriptObject;
     astore 2     // store scope in slot 2
     aload 2
     invokedynamic get:x(Object;)I
     istore 3
     iload 3
     aload 2
     invokedynamic get:y(Object;)I
     invokedynamic imul(II)I
     ireturn
```

```
public static f(ScriptFunction;Object;)I
     aload 0
     invokevirtual ScriptFunction.getScope()ScriptObject;
     astore 2     // store scope in slot 2
     aload 2
     invokedynamic get:x(Object;)I   // program point 1
     istore 3
     iload 3
     aload 2
     invokedynamic get:y(Object;)I   // program point 2
     invokedynamic imul(II)I         // program point 3
     ireturn
```

```
public static f(ScriptFunction;Object;)I
     aload 0
     invokevirtual ScriptFunction.getScope()ScriptObject;
     astore 2
     aload 2
     invokedynamic get:x(Object;)I    // program point 1
     istore 3
     iload 3
     aload 2
     invokedynamic get:y(Object;)I    // program point 2
     invokedynamic imul(II)I          // program point 3
     ireturn

     // catch (UnwarrantedOptimismException e)
     iconst_4
     anewarray Object    // 4 local variables in frame
     iload 3             // load variable 3 (tmp = x)
     invokedynamic populateArray([Object;I)[Object;
     goto rwe
     iconst_3
     anewarray Object    // 3 local variables in frame
rwe: aload 0             // load function
     aload 1             // load 'this'
     aload 2             // load scope
     invokedynamic populateArray([Object;Object;Object;Object;)[Object;
     iconst_0
     invokestatic getString$array(I)[String;
     invokestatic RewriteException.create(UnwarrantedOptimismException;[Object;
[String;)RewriteException;
     athrow
```

GCD – Ideal

```
public static gcd(Object;II)I

   iload 1
   istore 3
   iload 2
   istore 4
   iload 3
   iload 4
   if_icmpeq L10
   iload 3
   iload 4
   if_icmple L12
   iload 3
   iload 4
   isub
   istore 3
   goto L14
   iload 4
   iload 3
   isub
   istore 4
   goto L9
   iload 3
   ireturn
```

GCD – vs Nashorn

```
public static gcd(Object;II)I

   iload 1
   istore 3
   iload 2
   istore 4
   iload 3
   iload 4
   if_icmpeq L10
   iload 3
   iload 4
   if_icmple L12
   iload 3
   iload 4
   isub
   istore 3
   goto L14
   iload 4
   iload 3
   isub
   istore 4
   goto L9
   iload 3
   ireturn
```

```
public static gcd(Object;II)I
try {
   iload 1
   istore 3
   iload 2
   istore 4
   iload 3
   iload 4
   if_icmpeq L10
   iload 3
   iload 4
   if_icmple L12
   iload 3
   iload 4
   invokedynamic isub(II)I    //indy because sub may overflow
   istore 3
   goto L14
   iload 4
   iload 3
   invokedynamic isub(II)I    //indy because sub may overflow
   istore 4
   goto L9
   iload 3
   ireturn
} catch (UnwarrantedOptimismException e) {
   //save state and throw RewriteException to linker
}
```

Static Compiler Enhancements

- Liveness

```
function liveness() {
    var x = 0;
    for (var i = 0;i < 10; i++) {
       x += i;
    }
    x += "test";
    return x;
}
```

What about local variables? The slot type is hard coded in the local variable table!

Static Compiler Enhancements

- Liveness

```
function liveness() {
    var x = 0;
    for (var i = 0;i < 10; i++) {
       x += i;
    }
    x += "test";
    return x;
}
```

```
public static liveness(LObject;)LObject;
    iconst_0
    invokestatic Integer.valueOf (I)Integer;
    astore 2
    dconst_0
    dstore 3
    goto L2
    aload 2
    invokestatic JSType.toNumber (LObject;)D
    dload 3
    dadd
    invokestatic Double.valueOf (D)Double;
    astore 2
    dload 3
    dconst_1
    dadd
    dstore 3
    dload 3
    ldc 10.0
    dcmpg
    iflt L3
    aload 2
    ldc "test"
    invokedynamic ADD:OOO_I(LObject;LObject;)Object
    astore 2
    aload 2
    areturn

    local x Ljava/lang/Object; L0 L6 2
    local i D L0 L6 3
```

Static Compiler Enhancements

- Liveness

```
function liveness() {
    var x = 0;
    for (var i = 0;i < 10; i++) {
       x += i;
    }
    x += "test";
    return x;
}
```

```
public static liveness(Object;)Object;
   iconst_0
   istore 1
   iconst_0
   istore 3
   iload 3
   bipush 10
   if_icmpge L10
   iload 1
   iload 3
   invokedynamic iadd(II)I
   istore 1
   iload 3
   dup
   iconst_1
   invokedynamic iadd(II)I
   istore 3
   pop
   goto L9
   iload 1
   invokestatic Integer.valueOf(I)Integer;
   ldc "test"
   invokestatic ADD(Object;Object;)Object;
   astore 2
   aload 2
   areturn
   // exception handler goes here...

   local x              L6   L13  1    I
   local x              L13  L2   2    Object;
   local i              L8   L2   3    I
```

Field Representation

Field Representation

- `jdk.nashorn.internal.runtime.ScriptObject`
  - Contains a `PropertyMap`
    - Shape
  - Contains a number of fields
    - Scope variables
  - Contains a growing array
    - Spills – also scope variables
  - and lots of other things, e.g. `ArrayData`

Field Representation

```
public abstract class ScriptObject
    implements PropertyAccess {
    protected Object[] spillObjects;

    // most guards are a ref comparison on map
    private PropertyMap map;
}

public JO$4 extends ScriptObject {
    private Object o0, o1, o2, o3;

}
```

Property Maps

```
PropertyMap
     AccessorProperty: red   (flags = 0) slot = 0
     AccessorProperty: green (flags = 0) slot = 1
     AccessorProperty: blue  (flags = 0) slot = 3
     SpillProperty:    timeStamp (flags = NON_MODIFIABLE), index = 0
```

- Immutable
- Copy on write when any aspect changes
- Used for reference guards for almost all `invokedynamic` sites

Field Representation

```
var scopeVar;

function f() {
   scopeVar &= 17;
}
```

Field Representation

```
var scopeVar;

function f() {
   scopeVar &= 17;
}
```

1. Load scope var (`getfield`), stored as Object, is a `java.lang.Integer`
2. Coerce to `int` (unbox from `java.lang.Integer`)
3. Execute logical and (iand)
4. Coerce result to object (box to `java.lang.Integer`)
5. `putfield` result

Field Representation

```
var sc

functi
   sco
}
```

1. Lo
   ja
2. Co
3. Ex
4. Coerce result to object (box to `java.lang.Integer`)
5. `putfield` result

Field Representation

```
public abstract class ScriptObject
    implements PropertyAccess {
    protected Object[] spillObjects;

    // most guards are a ref comparison on map
    private PropertyMap map;
}

public JO$4 extends ScriptObject {
    private Object o0, o1, o2, o3;

}
```

Field Representation

```
public abstract class ScriptObject
    implements PropertyAccess {
    protected Object[] spillObjects;
    protected long[]   spillPrimitives;
    // most guards are a ref comparison on map
    private PropertyMap map;
}

public JO$4 extends ScriptObject {
    private Object o0, o1, o2, o3;
    private long   j0, j1, j2, j3;
}
```

Field Representation

```
public abstract class ScriptObject
    implements PropertyAccess {
    protected Object[] spillObjects;
    protected long[]   spillPrimitives;
    // most guards are a ref comparison on map
    private PropertyMap map;
}

public JO$4 extends ScriptObject {
    private Object o0, o1, o2, o3;
    private long   j0, j1, j2, j3;
}
```

“EWW! Fat pointers”

Field Representation

Setters

```
obj.j = (long)x;  //int
obj.j = x;        //long
obj.j = Double.doubleToRawLongBits(x); //double
obj.o  = x;       //object
```

Getters

```
return (int)obj.j;   //int
return obj.j;        //long
return Double.longBitsToDouble(obj.j); //double
return obj.o;        //object
```

Property Maps and Types

```
PropertyMap
     AccessorProperty: red   (flags = 0) slot = 0
     AccessorProperty: green (flags = 0) slot = 1
     AccessorProperty: blue  (flags = 0) slot = 3
     SpillProperty:    timeStamp (flags = NON_MODIFIABLE), index = 0
```

- Immutable
- Copy on write when any aspect changes
- Used for reference guards for almost all `invokedynamic` sites

Property Maps and Types

```
PropertyMap
     AccessorProperty: red   (flags = 0) slot = 0, type = int
     AccessorProperty: green (flags = 0) slot = 1, type = int
     AccessorProperty: blue  (flags = 0) slot = 3, type = int
     SpillProperty:    timeStamp (flags = NON_MODIFIABLE), index = 0,
type = object
```

- Immutable
- Copy on write when any aspect changes
  - INCLUDING FIELD TYPE
- Used for reference guards for almost all `invokedynamic` sites
  - INCLUDING FIELD GETTERS AND SETTERS

Field Representation

```
var scopeVar;

function f() {
   scopeVar &= 17;
}
```

1. Load scope var (`getfield`), stored as `java.lang.Integer`
2. Coerce to `int` (unbox from `java.lang.Integer`)
3. Execute logical and (iand)
4. Coerce result to object (box to `java.lang.Integer`)
5. `putfield` result

Field Representation

```
var scopeVar;

function f() {
   scopeVar &= 17;
}
```

1. Load scope var (`getfield`), stored as `long`
1. Coerce to `int` (l2i)
2. Execute logical and (iand)
3. Coerce to long (i2l)
4. `putfield` result

Field Representation

- Better representation?
  - Unsafe?
  - Needs some JVM magic, anyway
  - Test implementation: `sun.misc.TaggedArray`
  - Object shape change on GC?
  - Let's form an expert group!

Closing the circle: Optimistic Builtins

Optimistic builtins

```
/**
 * ECMA 15.4.4.7 Array.prototype.push (args...)
 *
 * @param self self reference
 * @param args arguments to push
 * @return array length after pushes
 */
@Function(attributes = Attribute.NOT_ENUMERABLE, arity = 1)
public static Object push(final Object self, final Object... args) {
    try {
        final ScriptObject sobj = (ScriptObject)self;

        if (bulkable(sobj) && sobj.getArray().length() + args.length <= JSType.MAX_UINT) {
            final ArrayData newData = sobj.getArray().push(true, args);
            sobj.setArray(newData);
            return newData.length();
        }

        long len = JSType.toUint32(sobj.getLength());
        for (final Object element : args) {
            sobj.set(len++, element, true);
        }
        sobj.set("length", len, true);

        return len;
    } catch (final ClassCastException | NullPointerException e) {
        throw typeError(Context.getGlobal(), e, "not.an.object", ScriptRuntime.safeToString(self));
    }
}
```

Relinking Callsites

- Can now be done with Exceptions as well as guards or SwitchPoints
- For example: continuous arrays
  - Previously done with multiple guards
- `array[x] & 17`
- Array getter links to:

```
try {
    //combinator - not real code
    return ((ContinuousArrayData)scriptObject.
        getArrayData()).
            getElementGetter(int.class).
                get(index); //turns AIOOBE to CCE
} catch (ClassCastException e) {
    //relink
}
```

Optimistic builtins

```
/**
 * ECMA 15.4.4.7 Array.prototype.push (args...)
 *
 * @param self self reference
 * @param args arguments to push
 * @return array length after pushes
 */
@Function(attributes = Attribute.NOT_ENUMERABLE, arity = 1)
public static Object push(final Object self, final Object... args) {
    try {
        final ScriptObject sobj = (ScriptObject)self;

        if (bulkable(sobj) && sobj.getArray().length() + args.length <= JSType.MAX_UINT) {
            final ArrayData newData = sobj.getArray().push(true, args);
            sobj.setArray(newData);
            return newData.length();
        }

        long len = JSType.toUint32(sobj.getLength());
        for (final Object element : args) {
            sobj.set(len++, element, true);
        }
        sobj.set("length", len, true);

        return len;
    } catch (final ClassCastException | NullPointerException e) {
        throw typeError(Context.getGlobal(), e, "not.an.object", ScriptRuntime.safeToString(self));
    }
}
```

Optimistic builtins

```
// pre: linker checks that callsite argument matches

@SpecializedFunction(guard=SpecializedFunction.Guard.CLASSCAST_EXCEPTION_CONTINUOUS_ARRAY)
public static long push(final Object self, final int arg) {
    try {
        return ((ContinuousArrayData)self).getArray()).push(arg);
    } catch (final NullPointerException e) {
        //fallthru
    }
    throw new ClassCastException();
}
```

Optimistic builtins

- Lots of candidates
  - `Array`
    - `push, pop, shift, unshift, length`
  - `String`
    - `charAt, charCodeAt, length`
  - `Number, Math`
  - `Function.apply` to `Function.call`
  - …
- We can use the same recompilation framework for native methods as for JavaScript methods

Antipattern that JS programmers love

```
var Class = {
  create: function() {
    return function() { //vararg
        this.initialize.apply(this, arguments);
    }
  }
};
Color = Class.create();

Color.prototype = {
    red: 0, green: 0, blue: 0,

    initialize: function(r, g, b) {
        this.red   = r;
        this.green = g;
        this.blue  = b;
    },
    toString: function() {
        return this.red   + " "
            + this.green + " "
            + this.blue;
    }
}
print(new Color(0xff, 0xee, 0xcc));
```

Other optimizations

Partial Evaluation

```
for (var i = 0; i < x.length; i++) {
    //x is loop invariant
)
```

- `x` can use `MethodHandle.constant` as an indy getter.
- We can use a `SwitchPoint` to invalidate it, given that `x` is modified in the scope
- Either forbid this callsite from being constant again, or allow `n` retries
- Or try with a receiver guard

Utilize `java.nio.ByteBuffer`

- Anyone using WebGL/ECMA 6 TypedArrays can get near-native performance
- `java.nio.ByteBuffer`
- 5-10x microbenchmark improvements
- Exported `BufferArray` to avatar.js

JVM / JDK optimizations

- Better type checking
- Math intrinsics
- Inlining modification
- Experiments with partial escape analysis / boxing removal
- Uncommon trap placement
- `java.lang.invoke`
  - Had to significantly beef up `MethodHandles.catchException` and many more
- …

There’s SO MUCH MORE I WANT TO TELL YOU

The cost of all this: *warmup*

Warmup

- The need to recompile methods has increased warmup time
- Tricks
  - `RewriteException` contains scope
    - More types are known when recompiling
  - Heuristics
  - Lazy compilation

Compiles gcd

```
function gcd(x, y) {
    var a = x;
    var b = y;
    while (a != b) {
       if (a > b) {
          a = a - b;
       } else {
          b = b - a;
       }
    }
    return a;
}

gcd(x, y);
```

Compiles gcd

```
function gcd(x, y) {
    var a = x;
    var b = y;
    while (a != b) {
       if (a > b) {
          a = a - b;
       } else {
          b = b - a;
       }
    }
    return a;
}

gcd(x, y);
```

Does not compile gcd

```
function gcd(x, y) {
    var a = x;
    var b = y;
    while (a != b) {
       if (a > b) {
          a = a - b;
       } else {
          b = b - a;
       }
    }
    return a;
}
```

Lazy Compilation

- Methods are only compiled on demand
- At link time
  - if no matching signature exists, compile one, as specific as possible
  - If matching signature does exist, try to compile an even more specific one
- Code quality is the same as eager compilation
  - Scope depths are maintained

Warmup

- Tricks (for run 2)
  - Persistent code cache (`--persistent-code-cache`)
  - Optimistic type information also cached to disk
  - TODO: parsed script for partial recompilation
- Reduces the second run to a very fast startup
  - … not the first run, however

The Future

Oracle JRockit Mission Control

File Window Help

flightRecording6643946283592296212.jfr

General
Memory
Code
CPU/Threads
WebLogic
Events

Servlets

Events Operative Set Synchronize Selection

Recording Start: 08:36:22 Recording End: 08:51:21

Servlet Operations

Shows servlet related information reported by the WebLogic Diagnostics Framework (WLDF).

Operation	Sample Count	Average Duration	Pressure
Servlet Request Run	41	237,870 ms	42,55%
Servlet Invocation	41	226,506 ms	40,52%
Servlet Execute	143	26,921 ms	16,80%
Servlet Response Send	41	495,781 µs	0,09%
Servlet Response Write Headers	44	260,959 µs	0,05%

Servlets Invocations by URI

Filter Column URI

URI	Sample Count	Average Duration	Pressure
/consolehelp/console-help.portal	3	2 s 707,148 ms	87,45%
/consolehelp/en-us/core/images/bullet.gif	3	58,796 ms	1,90%
/consolehelp/framework/skeletons/console/css/layout.css	1	135,351 ms	1,46%
/consolehelp/framework/skins/console/images/pageIdle.gif	1	93,835 ms	1,01%
/consolehelp/css/forms.css	1	58,173 ms	0,63%
/consolehelp/framework/skeletons/console/js/util.js	1	56,813 ms	0,61%
/consolehelp/framework/skins/console/css/menu.css	1	55,836 ms	0,60%

Overview | ECIDs | Users | Connector | Database | EJBs | JMS | JTA | Servlets | Web Services | HTTP Requests

Profiling and Java Flight Recorder

- A JFR “Nashorn” button
- JFR event production is simple, low overhead, and doesn’t require closing the source base
- Unique possibility to create a JavaScript profiler that no one else can do

Profiling and Java Flight Recorder

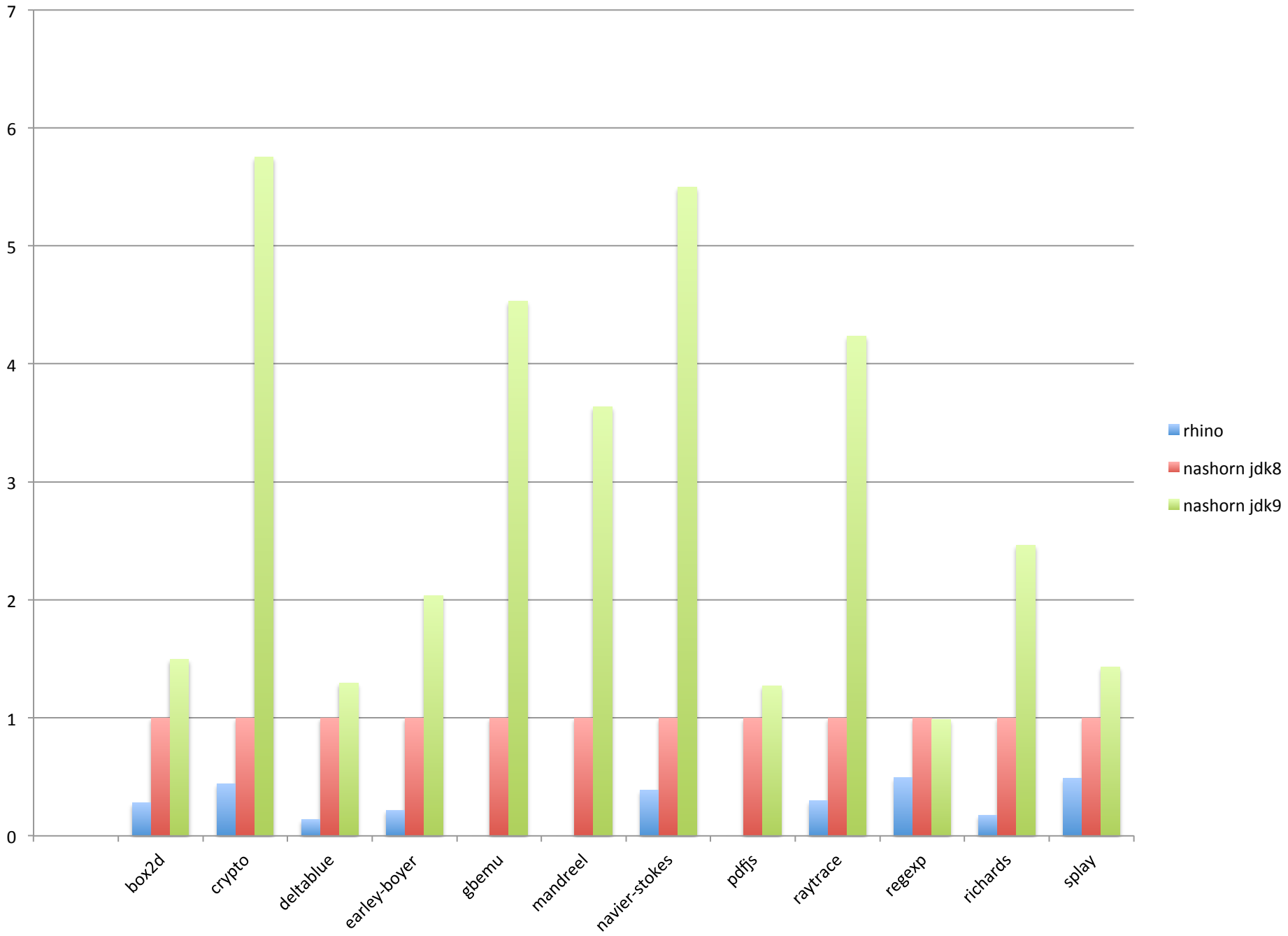
- A Nashorn JavaScript method is bytecode
- It has local variable tables, names, line numbers
- Fits 100% into the JFR framework already
- (IDE plugin for source level mapping)
- I already use JFR every day with this

JFR events – the sky's the limit

- Megamorphic callsite
- Suboptimal object shape
- Callsite hits, misses, values
- Scope access
- Fast scope access
- `with` entry/exit, `eval`, antipatterns
- Split methods
- Exceptions
- Relink with relink reason
- Assumption made
- Assumption invalidated
- Code pipeline reuse
- Timing
- Specialized method invocation
- Type widening
- Method invalidation
- Boxing / unboxing
- Non-trivial callsites
- Use of `arguments`
- Use of `callee`
- Code splitting
- Class creation
- Versions of compiled functions
- Code cache store/reuse
- Array meta information
- Compilation time
- Slow source lines
- Byte code size
- IR size histograms
- Index access on non-arrays
- Non-index access on arrays
- …

The Future

- Avatar
- the JVM
- the JDK
- Nashorn
- the Hardware (?)


7
6
5
4
3
2
1
0
box2d
crypto
deltablue
earley-boyer
gbemu
mandreel
navier-stokes
pdfjs
raytrace
regexp
richards
splay
rhino
nashorn jdk8
nashorn jdk9

The Future

Let Nashorn be the intelligent dynamic language execution framework for the JVM, regardless of frontend.

WORLD DOMINATION!

Q & A?

Give us your benchmarks, your experiments, your hacks,
your patch sets, we’d love your help!

Extra Slides after this point

Relinking Callsites

- An indy callsite is a subclass of `GuardedInvocation`
- Invalidation mechanisms:
  - Optional guard
  - Optional `SwitchPoint`

Relinking Callsites

- An indy callsite is a subclass of `GuardedInvocation`
- Invalidation mechanisms:
  - Optional guard
  - Optional `SwitchPoint`s (0 … n)
  - Optional Exception
    - Typically `ClassCastException`

```java
    /**
     * Constructor for a rewrite exception thrown from an optimistic function.
     *
     * @param e the {@link UnwarrantedOptimismException} that triggered this exception.
     * @param byteCodeSlots contents of local variable slots at the time of rewrite at the program point
     * @param byteCodeSymbolNames the names of the variables in the {@code byteCodeSlots} parameter.
     *
     * The array might have less elements, and some elements might be unnamed (the name can be null). The
     * information is provided in an effort to assist evaluation of expressions for their types by the
     * compiler doing the deoptimizing recompilation,
     * and can thus be incomplete - the more complete it is, the more expressions can be evaluated by the
     * compiler, and the more unnecessary deoptimizing compilations can be avoided.
     *
     * @return a new rewrite exception
     */

public static RewriteException create(final UnwarrantedOptimismException e,
            final Object[] byteCodeSlots,
            final String[] byteCodeSymbolNames);
```

Linker generates "restOf" method, which is identical but takes only RewriteException as argument, rewrites the frame state and jumps to the corresponding execution point until "restOf" method has executed. Old method is invalidated. Next call will generate a wider version.

apply to call

```
return function() { //vararg
    this.initialize.apply(this, arguments);
}

public static L:3(ScriptFunction;Object;[Object;)Object;
    aload 2  // args array
    aload 0  // callee
    iconst_0 // actual number of declared params
    invokestatic Global.allocateArguments([Object;Object;I)[Object;
    astore 3
    aload 1
    invokedynamic get:initialize(Object;)Object;
    dup
    invokedynamic get:apply(Object;)Object;
    swap
    aload 1
    aload 3
    invokedynamic dyn:call(Object;Object;Object;Object;)Object;
    pop
    getstatic ScriptRuntime.UNDEFINED
    areturn
```

apply to call

```
return function() { //vararg
    this.initialize.apply(this, arguments);
}

public static L:3(ScriptFunction;Object;III)Object;
    aload 1
    invokedynamic get:initialize(Object;)Object;
    dup
    invokedynamic get:apply(Object;)Object; //really loads “call”
    swap
    aload 1
    iload 2
    iload 3
    iload 4
    invokedynamic dyn:call(Object;Object;Object;III)Object;
    pop
    getstatic ScriptRuntime.UNDEFINED;
    areturn
```

apply to call

```
return function() { //vararg
    this.initialize.apply(this, arguments);
}

public static L:3(ScriptFunction;Object;III)Object;
    aload 1
    invokedynamic get:initialize(Object;)Object;
    dup
    invokedynamic get:apply(Object;)Object; //really loads “call”
    swap
    aload 1
    iload 2
    iload 3
    iload 4
    invokedynamic dyn:call(Object;Object;Object;III)Object;
    pop
    getstatic ScriptRuntime.UNDEFINED;
    areturn
```

```
    local :callee        L0   L1   0    ScriptFunction;
    local :this          L0   L1   1    Object;
    local :xarg0         L0   L1   2    I
    local :xarg1         L0   L1   3    I
    local :xarg2         L0   L1   4    I
```

apply to call

```
Function.prototype.call = function() {
   throw “This is a JavaScript! No performance for you!”;
}
```

SwitchPoint based dynamic invalidation – same mechanism as everewhere else