Implementing and Optimizing Dynamic Languages on the JVM

Marcus Lagergren
"Runtime Futurist"
Oracle
@lagergren

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?

- Computer scientist
- JRockit founder
  - Acquired by BEA, acquired by Oracle. Not likely to be acquired again.
  - Currently in the Java language team
- Low level guy
  - Compiler architect, virtualization OS hacker, hardware stuff
- High level guy
  - Tech evangelism, member of various program committees, supervisor of thesis students etc.
- Should sleep more

Agenda

- Background
- `invokedynamic` bytecodes and having the JVM do something fast with them
- Dynamic languages on the JVM
  - How to implement them
- The Nashorn Project
- Future directions (also in the JVM)
- Follow my struggle on Twitter: @lagergren

What do I Want?

Show you that dynamic languages are indeed feasible to implement on top of the JVM.

What do I Want?

No really, that is all ;-)

What to take with you from this talk

Abstract and main message

- Sell the JVM as a multi language platform
- The runtime gets you a lot for free
  - Memory Management
  - Code Optimizations
  - JSR-223 – Java Pluggability
- Performance
  - "Decent" and rapidly getting better in the near future

`invokedynamic` and `java.lang.invoke`

A new bytecode, the libraries around it and its applications

Invokedynamic

Introduction

- First time a new bytecode was introduced in the history of the JVM specification
- A new type of call
  - Previously: `invokestatic`, `invokevirtual`, `invoke-interface` and `invokespecial`.
- But more than that…

Invokedynamic

Introduction

- Along with its support framework, it may be roughly thought of as a function pointer
  - A way to do a call without the customary Java-language checks
  - Enables completely custom linkage
  - Essential if you want to hotswap method call targets
- Not something that `javac` will spit out
  - At least not currently. Lambdas will probably use it.
- First and foremost something you generate yourself when you weave bytecode for a dynamic language

`invokedynamic` bytecode

↓ calls

Bootstrap Method

↓ returns

`java.lang.invoke.CallSite`

↓ contains

Target (`java.lang.invoke.MethodHandle`)

Invokedynamic

java.lang.invoke.CallSite

- The concept of a CallSite
- One `invokedynamic` per CallSite
- Returned by the bootstrap call
- The holder for a a `MethodHandle`
  - The `MethodHandle` is the target
  - Target may be mutable or not
  - `getTarget` / `setTarget`

Invokedynamic

java.lang.invoke.CallSite

- The concept of a CallSite
- One `invokedynamic` per CallSite
- Returned by the bootstrap call
- The holder for a a `MethodHandle`
  - The `MethodHandle` is the target
  - Target may be mutable or not
  - `getTarget` / `setTarget`

```
 20: invokedynamic #97,0
// InvokeDynamic #0:"func":(Ljava/lang/Object;
    Ljava/lang/Object;)V

public static CallSite bootstrap(
   final MethodHandles.Lookup lookup,
   final String name,
   final MethodType type,
   Object… callsiteSpecificArgs) {

  MethodHandle target = f(
      name,
      callSiteSpecificArgs);
  // do stuff
  CallSite cs = new MutableCallSite(target);
  // do stuff

  return cs;
}
```

Invokedynamic

java.lang.invoke.MethodHandle

- MethodHandle concept:
- “This is your function pointer”

```
MethodType   mt = MethodType.methodType(String.class, char.class, char.class);
MethodHandle mh = lookup.findVirtual(String.class, "replace", mt);

String s = (String)mh.invokeExact("daddy", 'd', 'n');

assert "nanny".equals(s) : s;
```

Invokedynamic

java.lang.invoke.MethodHandle

- MethodHandle concept:
- "This is your function pointer"
- Logic may be woven into it
  - Guards: `c = if (guard) a(); else b();`
  - Parameter transforms/bindings

```
MethodHandle add =
   MethodHandles.guardWithTest(
      isInteger,
      addInt
      addDouble);
```

Invokedynamic

java.lang.invoke.MethodHandle

- MethodHandle concept:
- "This is your function pointer"
- Logic may be woven into it
  - Guards: `c = if (guard) a(); else b();`
  - Parameter transforms/bindings
- SwitchPoints
  - Function of 2 MethodHandles, `a` and `b`
  - Invalidation: rewrite CallSite `a` to `b`

```
MethodHandle add =
   MethodHandles.guardWithTest(
      isInteger,
      addInt
      addDouble);
```

```
SwitchPoint sp = new SwitchPoint();
MethodHandle add = sp.guardWithTest(
   addInt,
   addDouble);

// do stuff

if (notInts()) {
   sp.invalidate();
}
```

Invokedynamic

Performance of invokedynamic on the JVM

- What about performance?
- The JVM knows a callsite target and can inline it
  - No strange workaround machinery involved
  - Standard adaptive runtime assumptions, e.g. "guard taken"
- Superior performance
  - At least in theory
  - If you, for example, change CallSite targets too many times, you will certainly be punished for it by the JVM deoptimizing your code

Implementing Dynamic Languages on the JVM

Dynamic languages on the JVM

Hows and whys?

- I want to implement a dynamic language on the JVM
- Bytecode is already platform independent
- So what’s the problem?

Dynamic languages on the JVM

Hows and whys?

- I want to implement a dynamic language on the JVM
- Bytecode is already platform independent
- So what's the problem?
    - [don't get me started on bytecode]

Dynamic languages on the JVM

Hows and whys?

- I want to implement a dynamic language on the JVM
- Bytecode is already platform independent
- So what's the problem?
  - [don't get me started on bytecode]
  - Rewriting callsites – changing assumptions

Dynamic languages on the JVM

Hows and whys?

- I want to implement a dynamic language on the JVM
- Bytecode is already platform independent
- So what's the problem?
  - [don't get me started on bytecode]
  - Rewriting callsites – changing assumptions
  - But aside from that, the big problem is *types*!

Dynamic languages on the JVM

The problem with changing assumptions

- Assumptions may change at runtime to a much larger extent than typically is the case in a Java program
  - What? You deleted a field?
    - Then I need to change where this getter goes.
    - And all places who assume the object layout has more fields need to update

Dynamic languages on the JVM

The problem with changing assumptions

- Assumptions may change at runtime to a much larger extent than typically is the case in a Java program
  - What? You deleted a field?
    - Then I need to change where this getter goes.
    - And all places who assume the object layout has more fields need to update
  - What? You redefined `Math.sin` to always return `17`?

Dynamic languages on the JVM

The problem with changing assumptions

- Assumptions may change at runtime to a much larger extent than typically is the case in a Java program
  - What? You deleted a field?
    - Then I need to change where this getter goes.
    - And all places who assume the object layout has more fields need to update
  - What? You redefined `Math.sin` to always return `17`?
  - What? You set `func.constructor` to `3`? You are an idiot, but … OK then…

Dynamic languages on the JVM

The problem with weak types

- Consider this Java method

```
int sum(int a, int b) {
    return a + b;
}
```

Dynamic languages on the JVM

The problem with weak types

- Consider this Java method

```
int sum(int a, int b) {
    return a + b;
}
```

```
iload_1
iload_2
iadd
ireturn
```

- In Java, `int` types are known at compile time
- If you want to compile a `double` add, go somewhere else

Dynamic languages on the JVM

The problem with weak types

- Consider instead this JavaScript function

```
function sum(a, b) {
   return a + b;
}
```

Dynamic languages on the JVM

The problem with weak types

- Consider instead this JavaScript function

```
function sum(a, b) {
    return a + b;
}
```

```
???
???
???
???
```

- Not sure… `a` and `b` are something… that can be added.
- The + operator does a large number of horrible things.
    - Might even not commute if we are dealing with e.g. Strings here.

Dynamic languages on the JVM

ECMA 262 – The addition operator

11.6.1 The Addition operator (+)

The addition operator either performs string concatenation or numeric addition.

The production *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*). See the Note below 11.6.3.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native ECMAScript objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Host objects may handle the absence of a hint in some other manner.

NOTE 2 Step 7 differs from step 3 of the comparison algorithm for the relational operators (11.8.5), by using the logical-or operation instead of the logical-and operation.

Dynamic languages on the JVM

The problem with weak types

- Let's break it down a bit
- In JavaScript, `a` and `b` may start out as `ints` that fit in 32 bits
  - But the addition may overflow and turn the result into a `long`
  - … or a `double`
  - A JavaScript "number" is a somewhat fuzzy concept to the JVM
  - True for e.g. Ruby as well
- Type inference at compile time is way too weak

Dynamic languages on the JVM

GAMBLE!

- Remember the axiom of adaptive runtime behavior: GAMBLE!
  - The bad slow stuff probably doesn't happen
  - If we were wrong and it does, take the penalty THEN, not now.
- Pseudo Java – just a thought pattern

```
function sum(a, b) {
    try {
        int sum = (Integer)a + (Integer)b;
        checkIntOverflow(a, b, sum);
        return sum;
    } catch (OverFlowException | ClassCastException e) {
        return sumDoubles(a, b);
    }
}
```

Dynamic languages on the JVM

GAMBLE!

- Type specialization is the key
- The previous example was specialization without involving the Java 7+ mechanisms
- Even more generic:

```
final MethodHandle sumHandle = MethodHandles.guardWithTest(
   intsAndNotOverflow,
   sumInts,
   sumDoubles);

function sum(a, b) {
   return sumHandle(a, b);
}
```

Dynamic languages on the JVM

GAMBLE!

- We can use other mechanisms than guards too
  - Rewrite the target `MethodHandle` on `ClassCastException`
  - `SwitchPoints`
- Approach can be extended to Strings and other objects
- But the compile time types should be used if they ARE available
- Let's ignore integer overflows for now
  - Primitive number to object is another common scenario
  - Combine runtime analysis and invalidation with static types from the JavaScript compiler

Dynamic languages on the JVM

Add a pinch of static analysis

```
a = 4711.17;
b = 17.4711;
res *= sum(a, b);

//a, b known doubles
//result known double
```

Dynamic languages on the JVM

Add a pinch of static analysis

```
a = 4711.17;
b = 17.4711;
res *= sum(a, b);

//a, b known doubles
//result known double
```

```
//generic sum
sum(OO)O:
  aload_1
  aload_2
  invokestatic JSRuntime.add(OO)
  areturn
```

Dynamic languages on the JVM

Add a pinch of static analysis

```
a = 4711.17;
b = 17.4711;
res *= sum(a, b);

//a, b known doubles
//result known double
```

```
//generic sum
sum(OO)O:
  aload_1
  aload_2
  invokestatic JSRuntime.add(OO)
  areturn
```

```
ldc 4711.17
dstore 1
ldc 17.4711
dstore 2
dload 1
invoke JSRuntime.toObject(O)
dload 2
invoke JSRuntime.toObject(O)
invokedynamic sum(OO)O
invoke JSRuntime.toDouble(O)
dload 3
dmul
dstore 3
```

Dynamic languages on the JVM

Specialize the sum function for this callsite

- Doubles would still run faster than semantically equivalent objects
- Nice and short – just 4 bytecodes, no calls into the runtime

```
// specialized double sum
sum(DD)D:
  dload_1
  dload_2
  dadd
  dreturn
```

Dynamic languages on the JVM

But what if it's overwritten?

- In dynamic languages, anything can happen
- What if the program does this between callsite executions?

```
sum = function(a, b) {
  return a + 'string' + b;
}
```

- Use a SwitchPoint and generate a revert stub. Doesn't need to be explicit bytecode
- The CallSite will now point to the revert stub and not the double specialization

Dynamic languages on the JVM

```
sum(DD)D:
  dload_1
  dload_2
  dadd
  dreturn
```

```
sum_revert(DD)D: //hope this doesn't happen
  dload_1
  invokestatic JSRuntime.toObject(D)
  dload_2
  invokestatic JSRuntime.toObject(D)
  invokedynamic sum(OO)O
  invokestatic JSRuntime.toNumber(O)
  dreturn
```

None of the revert stub needs to be generated as actual explicit bytecode. MethodHandle combinators suffice.

Dynamic languages on the JVM

Result

```
ldc 4711.17
dstore 1
ldc 17.4711
dstore 2
dload 1
invoke JSRuntime.toObject(O)
dload 2
invoke JSRuntime.toObject(O)
invokedynamic sum(OO)O
invoke JSRuntime.toDouble(O)
dload 3
dmul
dstore 3
```

Dynamic languages on the JVM

Result

```
ldc 4711.17
dstore 1
ldc 17.4711
dstore 2
dload 1
invoke JSRuntime.toObject(O)
dload 2
invoke JSRuntime.toObject(O)
invokedynamic sum(OO)O
invoke JSRuntime.toDouble(O)
dload 3
dmul
dstore 3
```

```
ldc 4711.17
dstore 1
ldc 17.4711
dstore 2
dload 1

dload 2
//likely inlined:
invokedynamic sum(DD)D

dload 3
dmul
dstore 3
```

Dynamic languages on the JVM

Field Representation

- Assume types of variables don't change. If they do, they converge on a final type quickly
- Internal type representation can be a field, several fields or a "tagged value"
  - Reduce data bandwidth
  - Reduce boxing
- Remember *undefined*
  - Representation problems

```
var x;
print(x);        // getX()O

x = 17;          // setX(I)
print(x);        // getX()O

x *= 4711.17;    // setX(D)
print(x);        // getX()O

x += "string";   // setX(O)
print(x);        // getX()O
```

```
// naïve impl
// don't do this

class XObject {
   int xi;
   double xd;
   Object xo;
}
```

Dynamic languages on the JVM

Field Representation – getters on the fly – use SwitchPoints

- Not actual code – generated by MethodHandles

```
int getXWhenUndefined()I {
  return 0;
}
double getXWhenUndefined()D {
  return NaN;
}
Object getXWhenUndefined()O {
  return Undefined.UNDEFINED;
}
```

```
int getXWhenInt()I {
  return xi;
}
double getXWhenInt()D {
  return JSRuntime.toNumber(xi);
}
Object getXWhenInt()O {
  return JSRuntime.toObject(xi)
}
```

```
int getXWhenDouble()I {
  return JSRuntime.toInt32(xd);
}
double getXWhenDouble()D {
  return xd;
}
Object getXWhenDouble()O {
  return JSRuntime.toObj(xd);
}
```

```
int getXWhenObject()I {
  return JSRuntime.toInt32(xo);
}
double getXWhenObject()D {
  return JSRuntime.toNumber(xo);
}
Object getXWhenObject()O {
  return xo;
}
```

Dynamic languages on the JVM

Field Representation – setters

- Setters to a wider type T trigger all SwitchPoints up to that type

```
void setXWhenInt(int i) {
  this.xi = i; //we remain an int, wohooo!
}

void setXWhenInt(double d) {
  this.xd = d;
  SwitchPoint.invalidate(xToDouble);
  //invalidate next switchpoint, now a double;
}

void setXWhenInt(Object o) {
  this.xo = o;
  SwitchPoint.invalidate(xToDouble, xToObject)
  //invalidate all remaining switchpoints, now an Object forevermore.
}
```

The Nashorn Project

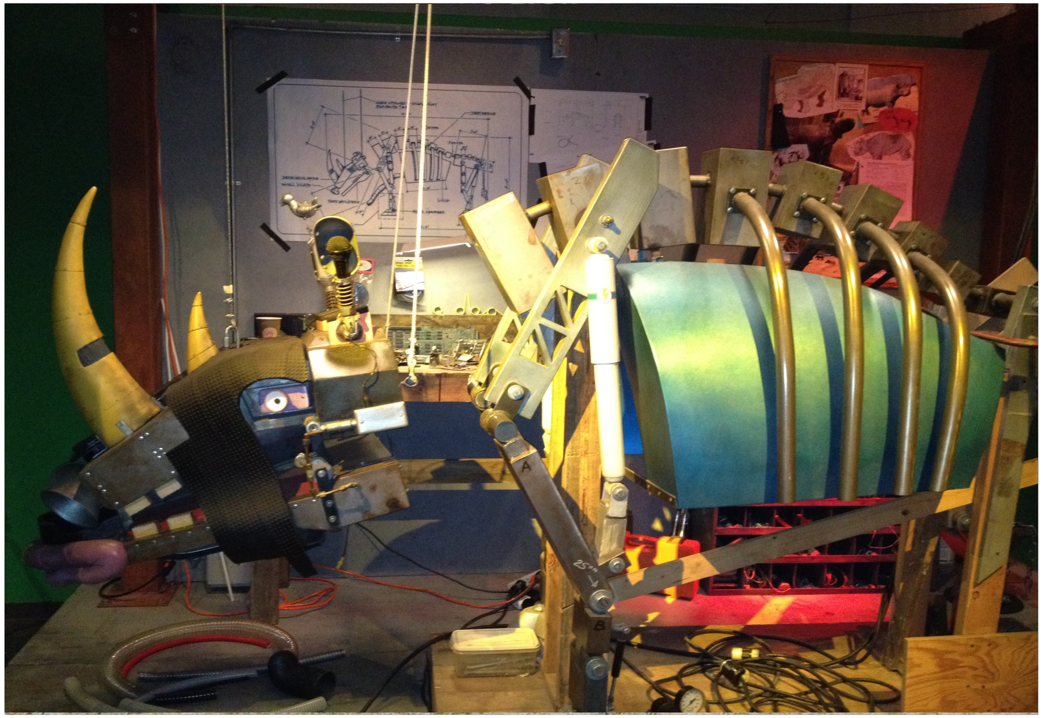
JavaScript using
`invokedynamic`

The Nashorn Project

- A Rhino for 2013 (aiming for open source release in the Java 8 timeframe)
- Nashorn is German for Rhino (also sounds cool)

The Nashorn Project

- A Rhino for 2013 (aiming for open source release in the Java 8 timeframe)
- Nashorn is German for Rhino (also sounds cool)



The Nashorn Project

Rationale

- Create a 100% pure Java `invokedynamic` based POC of a dynamic language implementation on top of the JVM
- It should be faster than any previous `invokedynamic`-free implementations
- Become the ultimate `invokedynamic` consumer, to make sure this stuff works
- Performance bottlenecks in the JVM should be cross communicated between teams

The Nashorn Project

Rationale

- JavaScript was chosen
  - Rhino, the only existing equivalent is slow
  - Rhino codebase contains all deprecated backwards compatibility ever
  - Ripe for replacement
- JSR-223 – Java to JavaScript, JavaScript to Java
  - Automatic support. Very powerful
- The JRuby folks are already doing an excellent work with JRuby

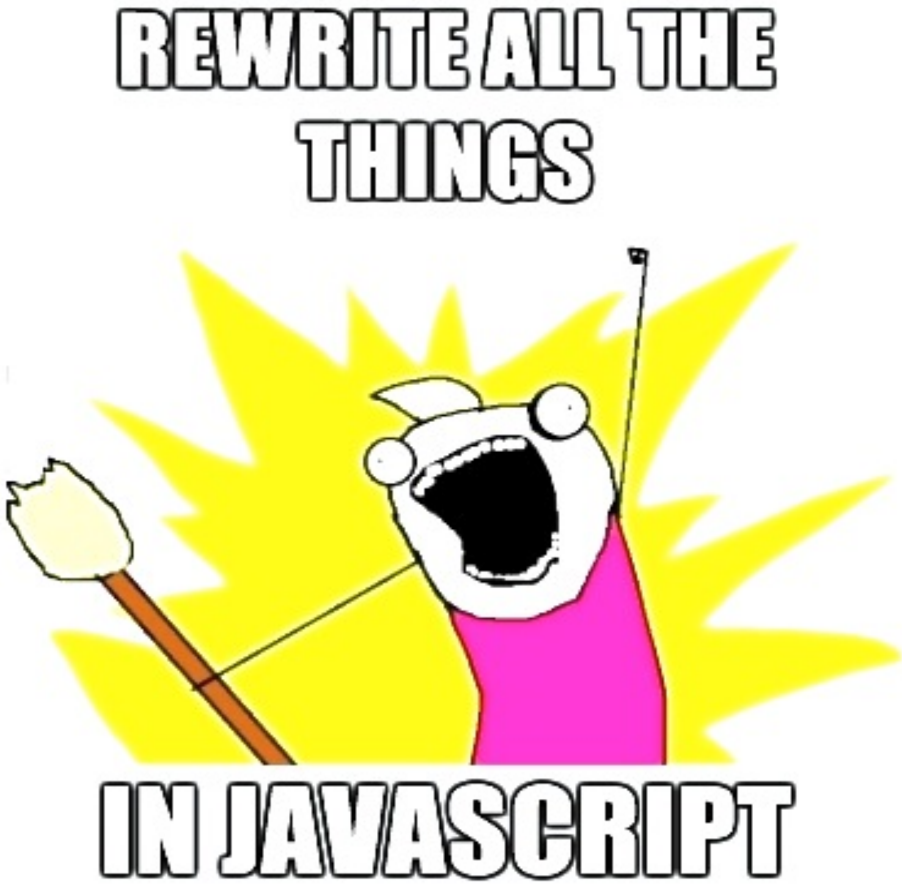
The real reason – Keep up with Atwood's law:

Atwood's law: *"Any application that can be written in JavaScript, will eventually be written in JavaScript"*

- James Atwood (founder, stackoverflow.com)

The real reason – Keep up with Atwood's law:

2nd law of Thermodynamics: *"In all closed systems, entropy must remain the same or increase"*


REWRITE ALL THE THINGS
IN JAVASCRIPT


ORACLE

The Nashorn Project

Rationale

- Do a node.js implementation that works with Nashorn
  - "node.jar" (Async I/O implemented with Grizzly)
- 4-5 people working fulltime in the langtools group.
- Nashorn is scheduled for open source release in the Java 8 timeframe
  - Source code is currently available in the OpenJDK repo
  - node.jar has no official schedule yet
- Other things that will go into the JDK
  - Dynalink (finalizing legal approval – hopefully there for M7)
  - ASM (already integrated into Java8)

The Nashorn Project

Challenge – JavaScript is an awful, horrible language

The Nashorn Project

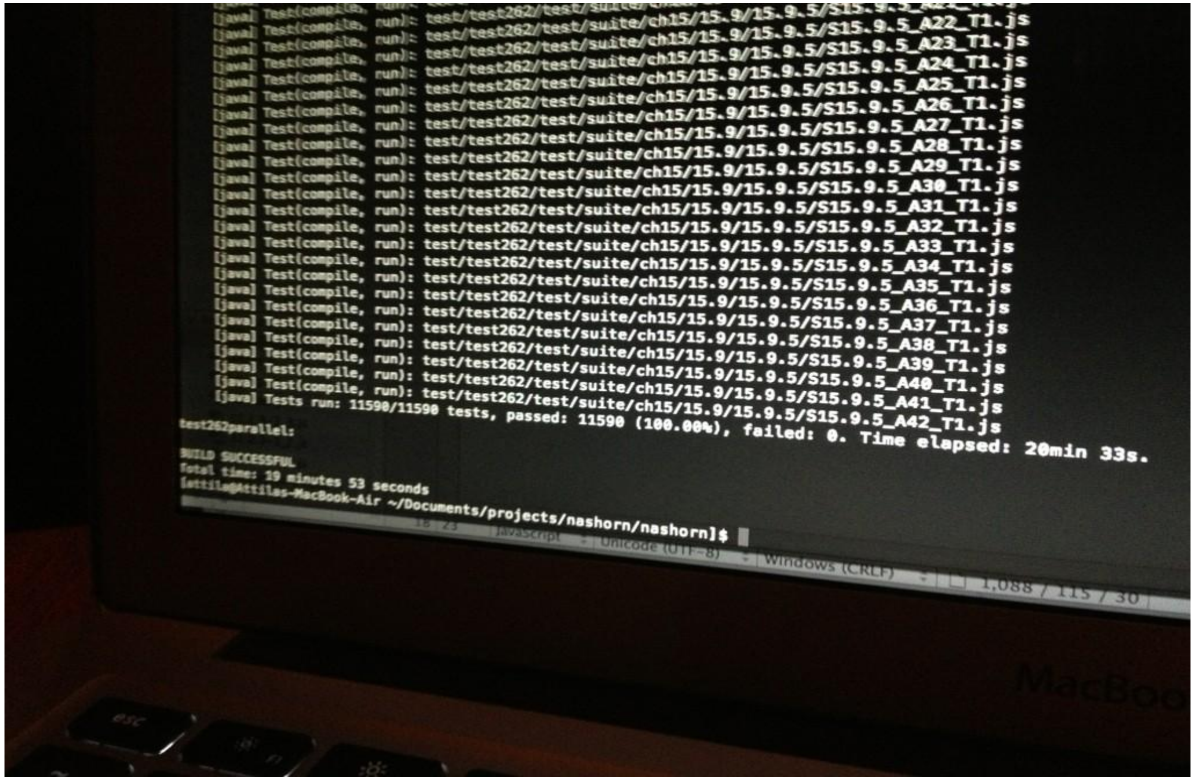
Challenge – JavaScript is an awful, horrible language

- `‘4’ - 2 === 2`, but `‘4’ + 2 === ’42’`
- Can I have variable declarations after their usages? Of course you can!
- The entire `with` keyword
- `Number(“0xffgarbage”) === 255`
- `Math.min() > Math.max() === true`
- `Array.prototype[1] = 17; var a = [,,,]; print(a) : [,17,]`
- So I take this floating point number and shift it right…
- `a.x` looks just like a field access
  - May just as easily be a getter with side effects (`a` too for that matter)
- `[] + {}, {} + [], [] + [], {} + {}`
- I could go on, but anyway, it’s a compiler/runtime writer’s worst nightmare

Compliance

Scene: a rainy fall evening at a pub in Stockholm. Attila (@asz) running the ECMA test suite [1]… ~11,500 tests…

[1] http://test262.ecmascript.org


[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A22_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A23_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A24_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A25_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A26_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A27_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A28_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A29_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A30_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A31_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A32_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A33_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A34_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A35_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A36_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A37_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A38_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A39_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A40_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A41_T1.js
[java] Test(compile, run): test/test262/test/suite/ch15/15.9/15.9.5/S15.9.5_A42_T1.js
[java] Tests run: 11590/11590 tests, passed: 11590 (100.00%), failed: 0. Time elapsed: 20min 33s.
test262parallel:
BUILD SUCCESSFUL
Total time: 19 minutes 53 seconds
[attila@Attilas-MacBook-Air ~/Documents/projects/nashorn/nashorn]$

100%! WOHOO!


ORACLE

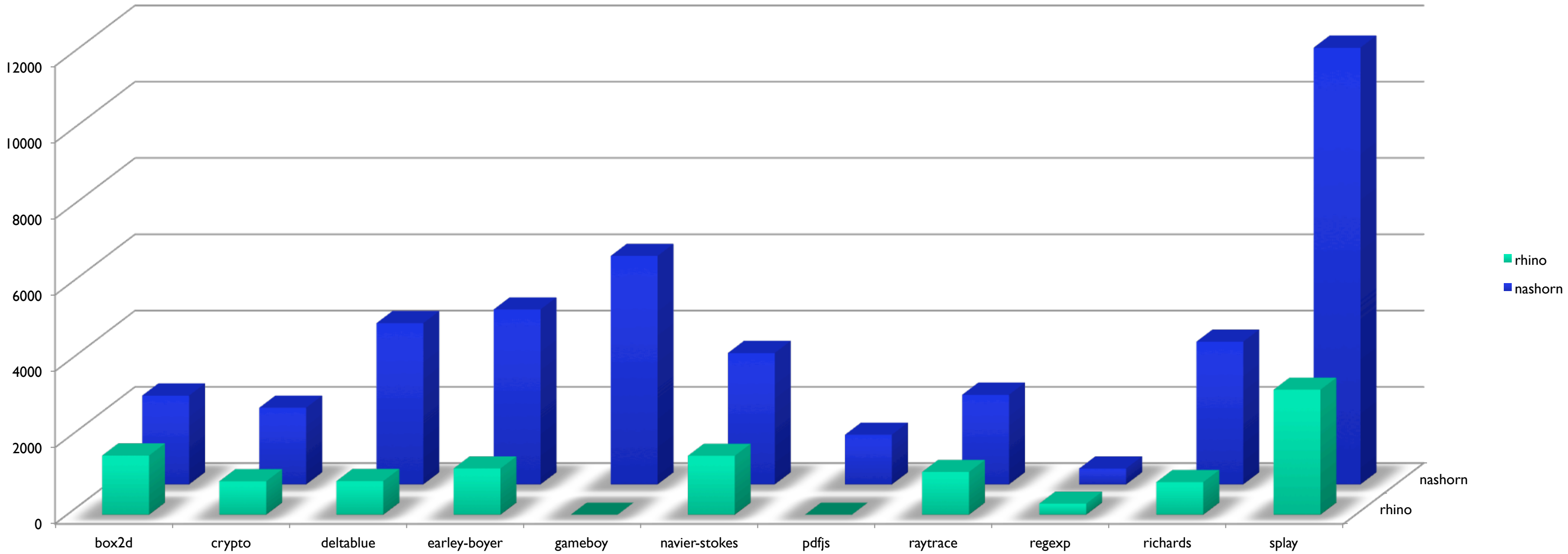
The Nashorn Project

Compliance

- At the time of writing we have full ECMAScript compliance
- This is better than ANY existing JavaScript runtime
- Rhino, somewhat surprisingly, is only at ~94%
- Shifting focus more and more towards performance…

The Nashorn Project

Performance



The Nashorn Project

So why not V8/Spidermonkey/other native runtime then?

- Nashorn is not a single threaded C++ monolith
- Nashorn is a lot smaller in scope as it does not need its own runtime
  - nashorn.jar is ~1MB
  - Project Jigsaw will help us even more
- Multithreading
- Free portability across hardware platforms
- Our node.jar implementation is already quite fast and much smaller than node.js

The Nashorn Project

So why not V8/Spidermonkey/other native runtime then?

- JSR-223
  - Powerful
  - Java can call JavaScript
  - JavaScript can call Java
- Makes things like node.jar significantly less complex
- You WANT this a JavaScript developer

```
import javax.script.*;

Object z = x.get(“y”);
x.put(“y”, z);
```

```
var random = new java.util.Random;

java.lang.System.out.println(random.nextInt());
```

```
var runnable = new java.lang.Runnable({
    run: function() { console.log('running'); }
});
var executor = java.util.concurrent.Executors.
    newCachedThreadPool();

executor.submit(runnable);
```

The Nashorn Project

So why not V8/Spidermonkey/other native runtime then?

- Killer apps? It is very attractive with a small self contained node.jar in the Java EE cloud as well as in embedded environments
  - We have successfully deployed Nashorn running node.jar on a Raspberri Pi board.
  - How cool is that? ;-)
- Java Mission Control!
- The future will bring further Nashorn AND JVM performance improvements.

The Nashorn Project

More info, please!

- `hg clone http://hg.openjdk.java.net/nashorn/jdk8/nashorn`
- `cd make ; make`
- Check the Nashorn blog for news
  - `http://blogs.oracle.com/nashorn`

Improvements on the Horizon

Nashorn performance. Invoke dynamic performance. JVM performance.

Charlie Nutter @FOSDEM: "Performance – I believe. I really do. But it has gone back and forth"

The Nashorn Project

Nashorn improvements

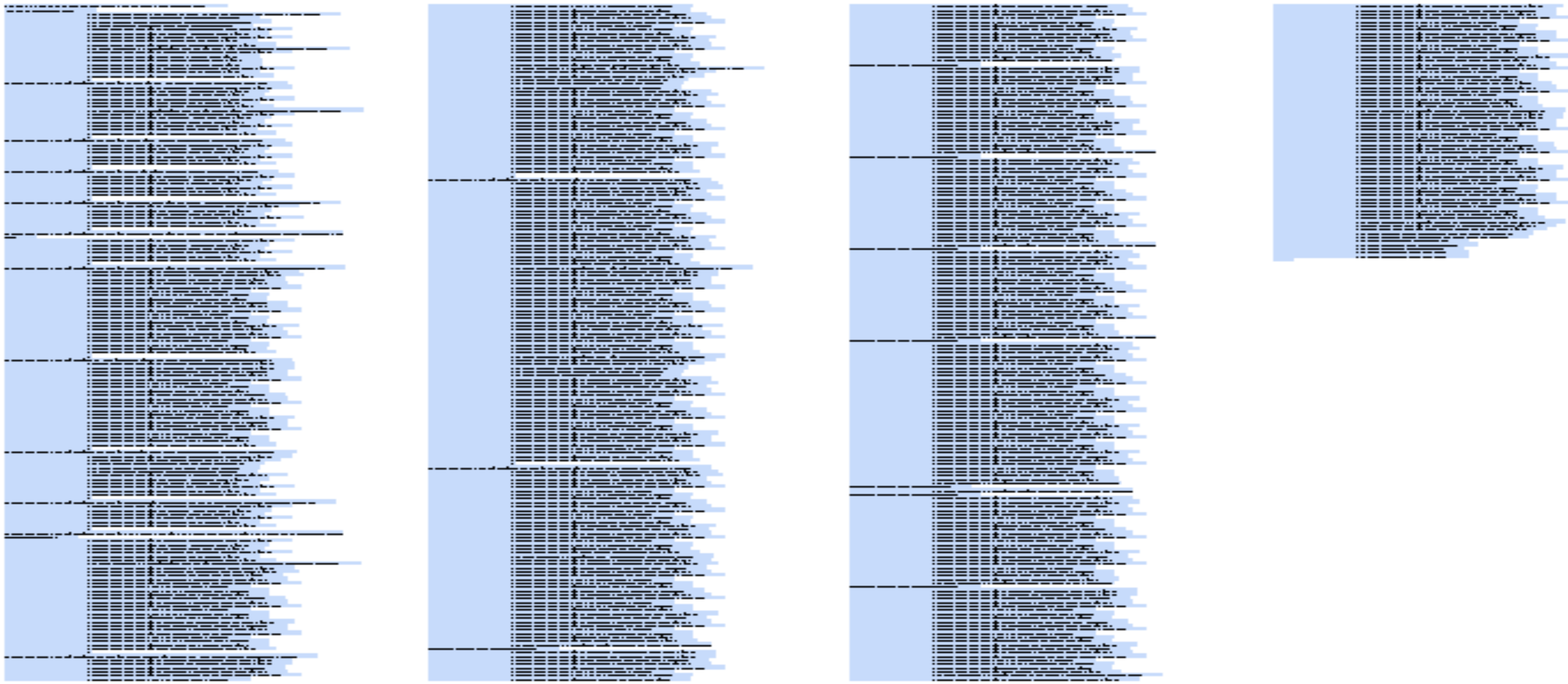
- Performance, performance, performance.
- Look at parallel APIs
- Lazy execution architecture
- Library improvements
  - RegExp
  - Possible integration with existing 3rd party solutions
- TaggedArrays – grope around a bit in the JVM internals
  - Not too much

JVM Improvements

JVM improvements

- Permgen removal
  - Classic problem with OOM generating lots of bytecode
- Stability
  - Java 7 Invokedynamic had stability issues
  - Java 8 MethodHandle framework rewritten mostly in Java
    - LambdaForms (entire MH chain in Java)
    - Going into upcoming 7 backport.

JVM Improvements



JVM Improvements

JVM improvements

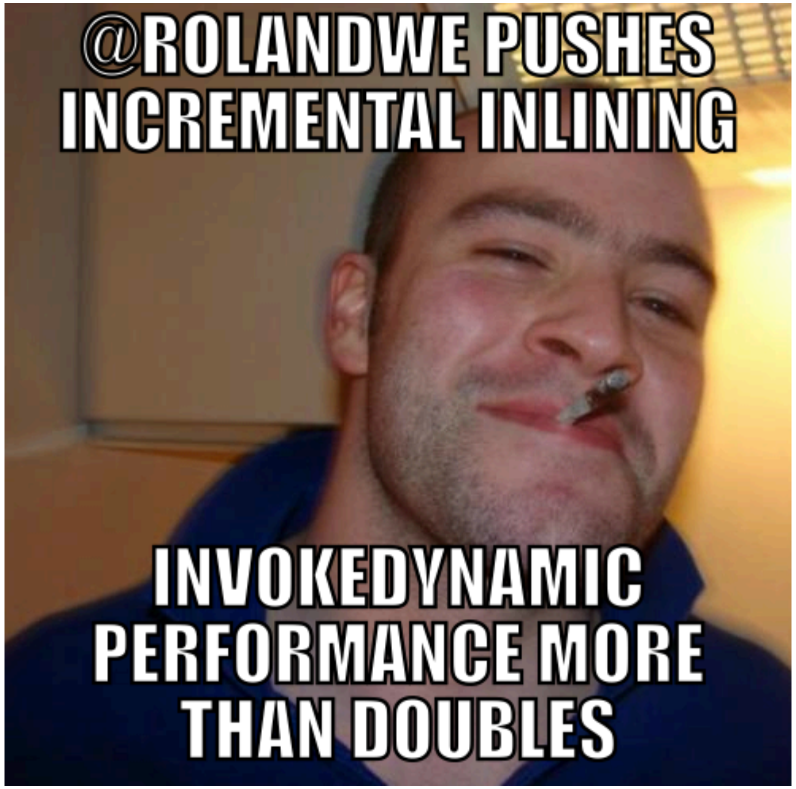
- Inlining artifacts matter a lot for callsites
  - Need incremental inlining
  - LambdaForms make stack traces huge. If we don't inline better we are dead.
  - Good inlining begets local escape analysis which begets boxing removal – boxing is our other enemy.

JVM Improvements

First I was like…

JVM Improvements

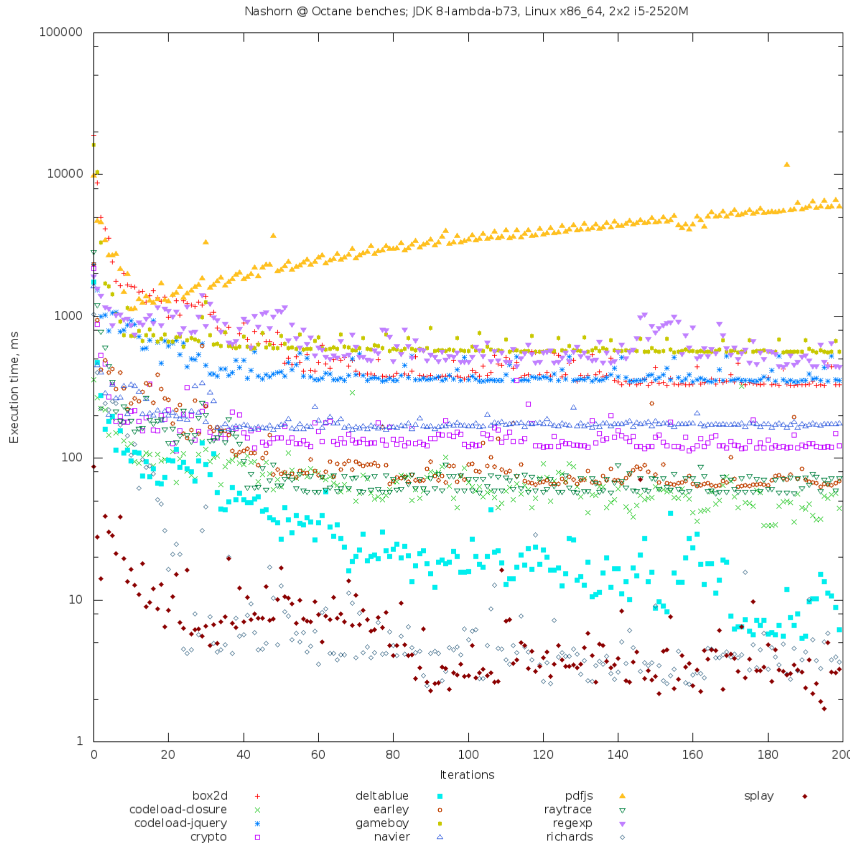
...but then...



JVM Improvements

LambdaForms

- LambdaForms
- What? A third JIT?
- Warmup issues
  - ..being addressed
  - Interpretor overhead



In conclusion

Open source!

- The good news: YOU CAN HELP!
- The Nashorn project: `hg clone http://hg.openjdk.java.net/nashorn/jdk8/nashorn`
- The Da Vinci Machine Project: `http://openjdk.java.net/projects/mlvm/`
- The open source plan is
  1. Ask the community to contribute *functionality, testing, performance, performance analysis, bug fixes, library optimizations, test runs with "real" applications, browser simulation frameworks, kick-ass hybrid Java solutions*
  2. …?
  3. Profit!

Follow me on Twitter: @lagergren

Q&A