

THE QUEST FOR DYNAMIC LANGUAGE PERFORMANCE ON THE JVM

[NASHORN WAR STORIES]

(from a battle scarred veteran of
`invokedynamic`)

Marcus Lagergren

Oracle

INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE

gotocon.com

[NASHORN RANTS]

Marcus Lagergren

Oracle

INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE

gotocon.com

[JAVASCRIPT RANTS]

Marcus Lagergren

Oracle

INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE

gotocon.com



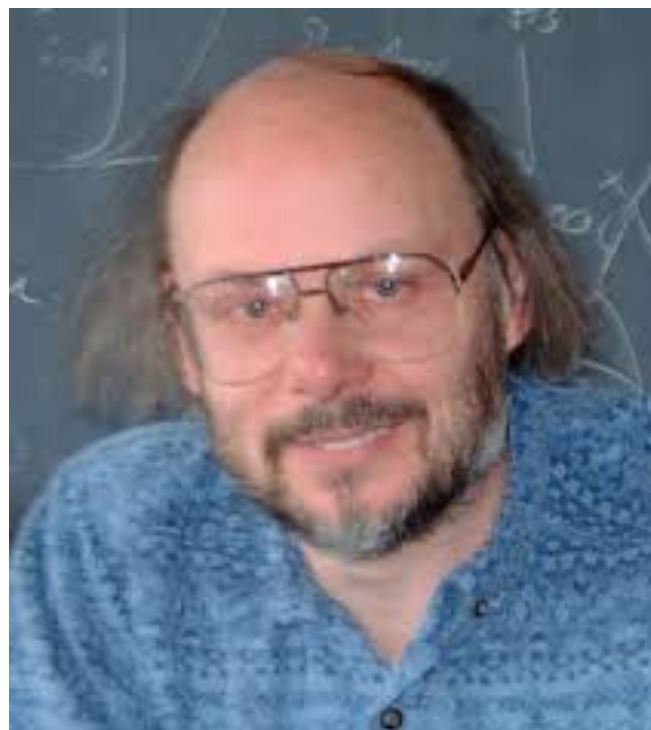
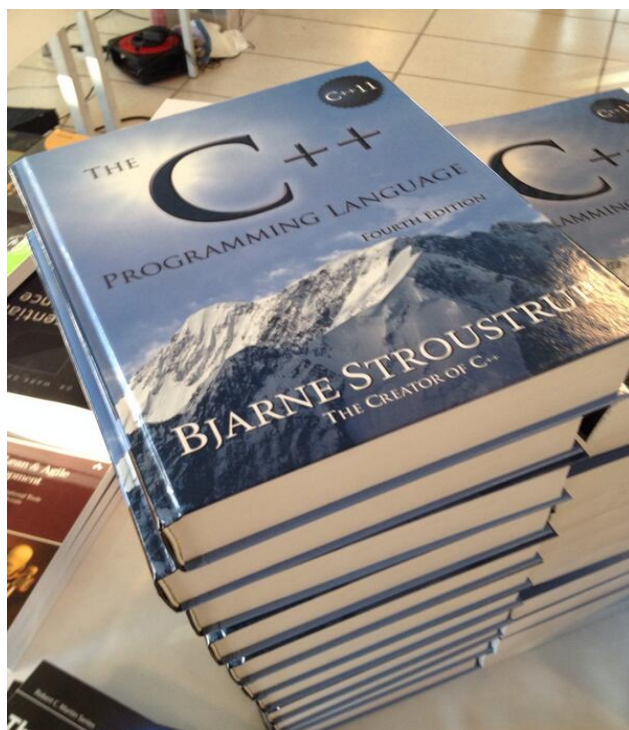
The Legal Slide

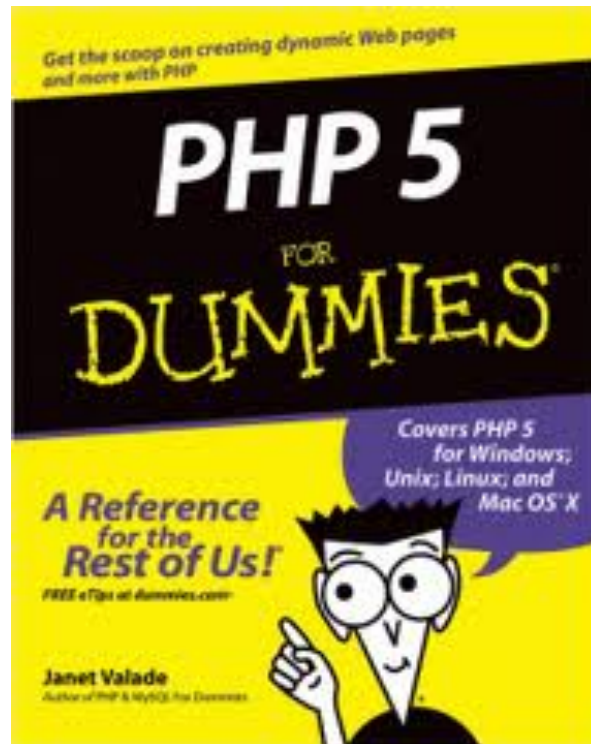
"THE FOLLOWING IS INTENDED TO OUTLINE OUR GENERAL PRODUCT DIRECTION. IT IS INTENDED FOR INFORMATION PURPOSES ONLY, AND MAY NOT BE INCORPORATED INTO ANY CONTRACT. IT IS NOT A COMMITMENT TO DELIVER ANY MATERIAL, CODE, OR FUNCTIONALITY, AND SHOULD NOT BE RELIED UPON IN MAKING PURCHASING DECISION. THE DEVELOPMENT, RELEASE, AND TIMING OF ANY FEATURES OR FUNCTIONALITY DESCRIBED FOR ORACLE'S PRODUCTS REMAINS AT THE SOLE DISCRETION OF ORACLE."

Who am I?



@lagergren











I am here to talk about...



I am here to talk about...

What we've suffered through so far to
implement a dynamic language on the JVM



I am here to talk about...

What we've suffered through so far to
implement a dynamic language on the JVM



The **Nashorn** Project



Also – a parade of JavaScript horrors



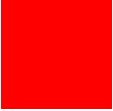


Agenda

- What is Nashorn and why?
- The problem of compiling an alien language to Java [sic] bytecode
 - Types
 - Optimistic assumptions
- The JVM and its issues

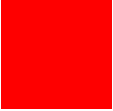


What is Nashorn and why?



What is Nashorn?

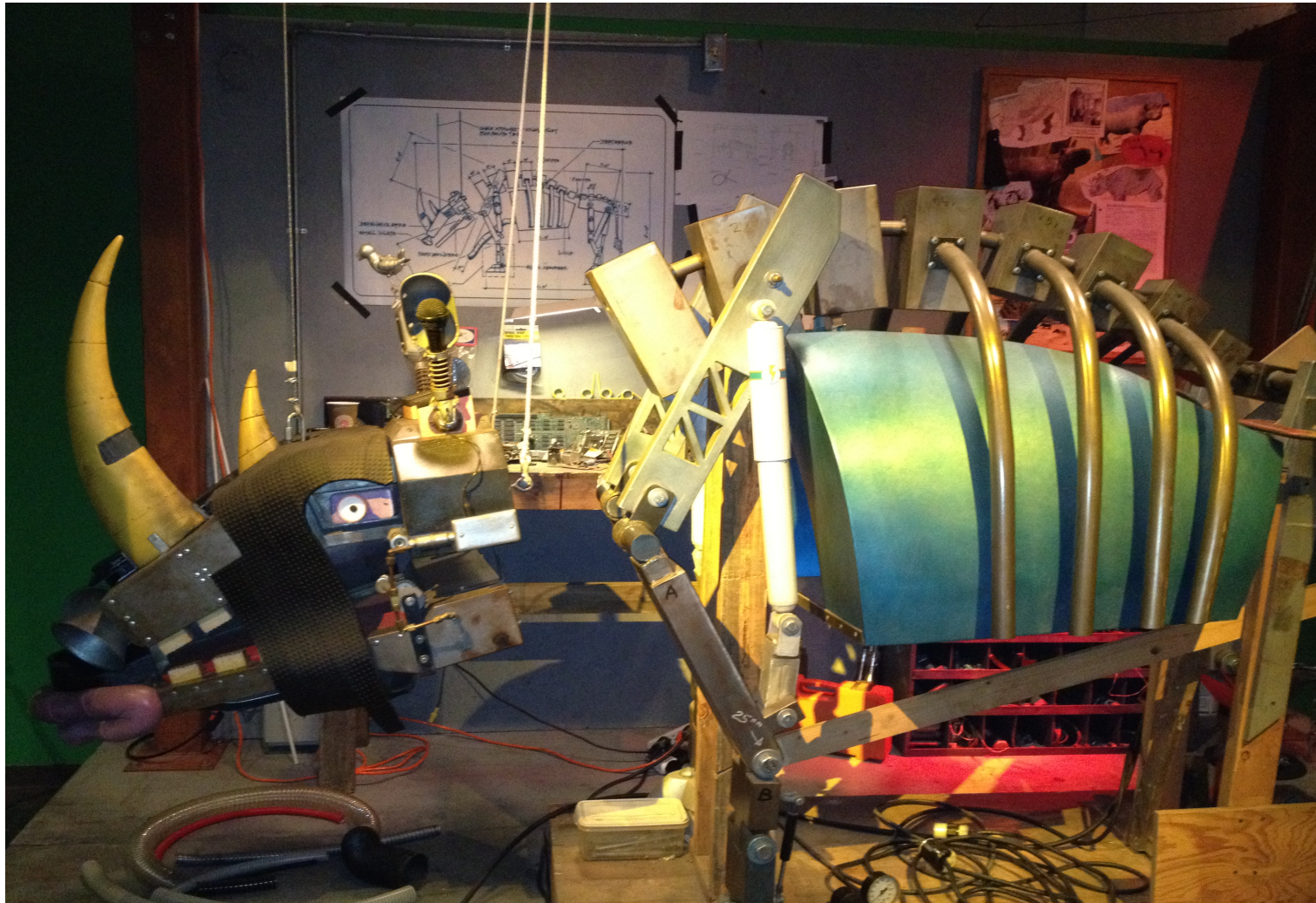
- Nashorn is a 100% pure Java runtime for JavaScript
- Nashorn generates bytecode
 - Invokedynamics are everywhere
- Nashorn currently performs somewhere on the order of ~2-10x better than Rhino
- Nashorn is in JDK 8
- Nashorn is 100% ECMAScript compliant
- Nashorn has a well thought through security model



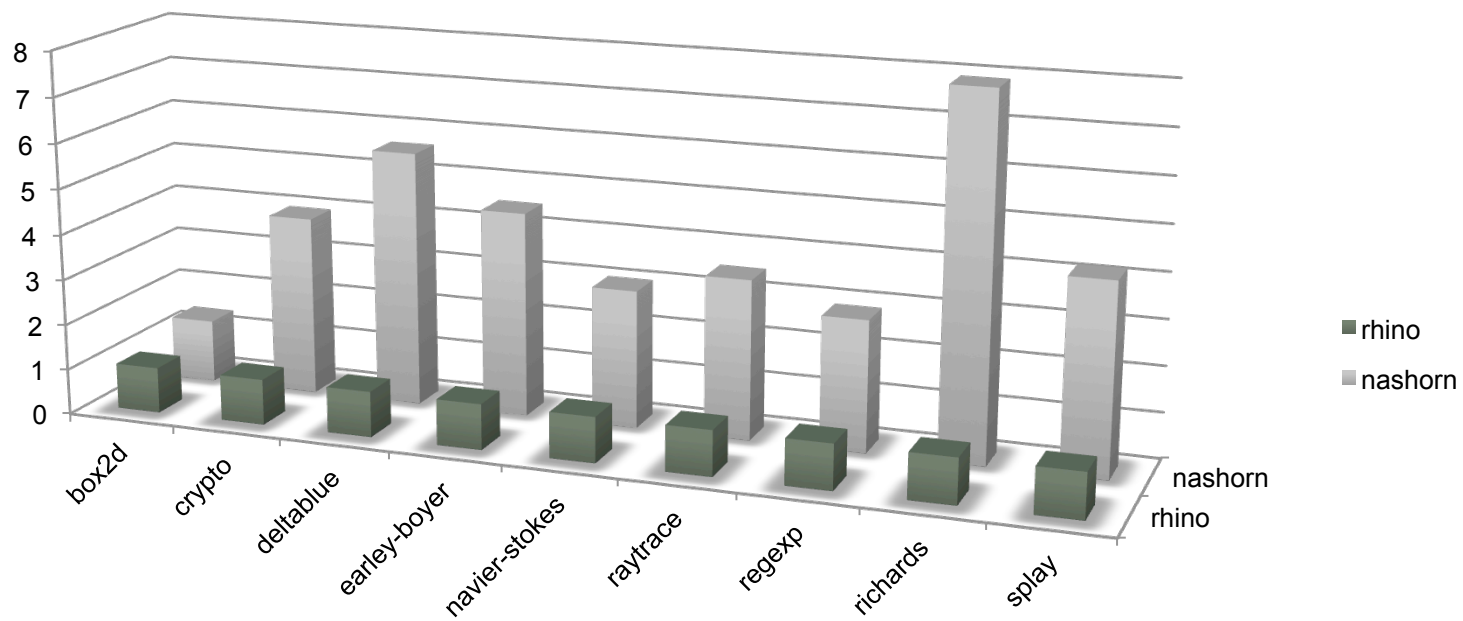
Why Nashorn?

- Started as an `invokedynamic` POC.
- Rhino is still alive today after ~18 years. Why?
 - JSR-223
- Nashorn is now mature and replaces Rhino for Java 8





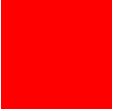
Performance





Performance

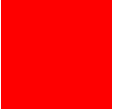




When is Nashorn available?

- Nashorn is part of OpenJDK8
 - Already available in JDK 8 builds.

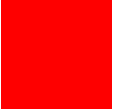
```
> jjs  
jjs> var x = "hello";  
jjs> print(x);  
hello  
jjs>
```



Compiling an alien language to Java [sic] bytecode

■ Compiling an alien (non-Java language) to bytecode





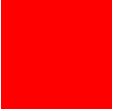
Compiling an alien (non-Java language) to bytecode

- Scala is fairly good fit



Compiling an alien (non-Java language) to bytecode

- Scala is fairly good fit
 - Yes I know: hard tail call optimization, interface injection etc.



Compiling an alien (non-Java language) to bytecode

- Scala is fairly good fit
 - Yes I know: hard tail call optimization, interface injection etc.
- Ruby and JavaScript are pretty bad fits



Compiling an alien (non-Java language) to bytecode

- Scala is fairly good fit
 - Yes I know: hard tail call optimization, interface injection etc.
- Ruby and JavaScript are pretty bad fits
 - No types
 - Things change at runtime. A lot.
 - Invokedynamic certainly alleviates a lot of the pain, but plenty of stuff remains to be solved



JavaScript!

Was it deliberately
designed to make every
efficient representation
useless?



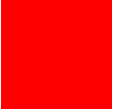
Let's talk about JavaScript

```
jjs> Array.prototype[1] = 17;
```



Let's talk about JavaScript

```
jjs> Array.prototype[1] = 17;  
17  
jjs>
```

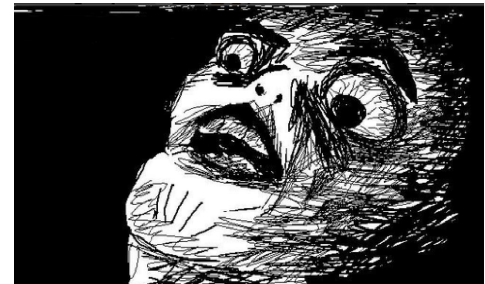


Let's talk about JavaScript

```
jjs> Array.prototype[1] = 17;  
17  
jjs> print([,,,]);
```

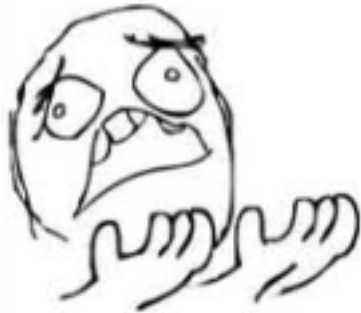
Let's talk about JavaScript

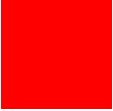
```
jjs> Array.prototype[1] = 17;  
17  
jjs> print([,,,]);  
,17,  
jjs>
```



Let's talk about JavaScript - Numbers

- Numbers in JavaScript have no fixed ranges
- “Intish”. “Doublish”.
- Not very nice for strongly typed bytecode
 - Overflows must be handled
- Conservative: At least they tend to fit in Java doubles.





Let's talk about JavaScript - Numbers

- Double arithmetic is slower than integer arithmetic on modern HW
- But double arithmetic is sometimes *faster* than int arithmetic with the necessary overflow checks.
- WAT!
- (getting back to that)



Let's talk about JavaScript – Types/Numbers

- HotSpot itself was originally tested and developed with bytecode that came from Java
- Representing everything as Objects to get the bytecode format type agnostic is nowhere near viable, performance wise.
- Boxing
- Go primitive

We should

- For bytecode performance we should
 - Use whatever static types we have
 - (mostly) done
 - Optimistically assume stuff about types
 - On it

CHALLENGE ACCEPTED





Let's talk about JavaScript – Static type info

- JavaScript type coercion semantics and literals – uses and definitions
- That's all the static type info we're going to get from the compiler
 - Java `int`: statically enough for `~`, `&`, `|`, `^`
 - Java `double`: statically enough for: `*`, `/`, `-`, `%`
 - Object: binary + and pretty much everything else



Let's talk about JavaScript – Static type info

- Callsites, though. How do we deal with parameter types?

```
int square(int x) {  
    return x * x;  
}
```

```
iload_0  
dup  
imul  
ireturn
```



Let's talk about JavaScript – Static type info

- But...

```
function square(x) {  
    return x * x;  
}
```

```
jjs> square(2)
```

```
4
```

```
jjs> square(2.1)
```

```
4.41
```

```
jjs> square("a")
```

```
NaN
```



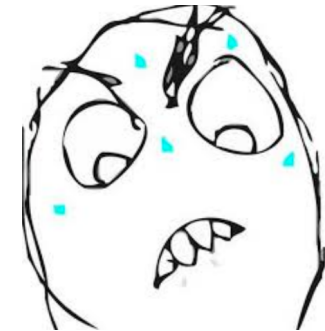
Let's talk about JavaScript – Static type info

- So conservatively...

```
square(Ljava/lang/Object;)D
  aload_0
  // hopefully just unbox:
  invokestatic coerce2Double(Ljava/lang/Object;)D
  dup
  dmul    // returns mul result, so always double
  dreturn
```

Let's talk about JavaScript – Static type info

- Guess again



```
jjs> square({  
    valueOf: function() {  
        global++;  
        return 2 + global; });  
  
...
```



Let's talk about JavaScript – Static type info

- So conservatively...

```
square(Ljava/lang/Object;)D
  aload_0
  // hopefully just unbox:
  invokestatic coerce2Double(Ljava/lang/Object;)D
  dup
  dmul    // returns mul result, so always double
  dreturn
```



Let's talk about JavaScript – Static type info

sigh - well at least the return value HAS to be double

```
square(Ljava/lang/Object;)D
  aload_0
  invokestatic coerce2Double(Ljava/lang/Object;)D
  aload_0
  invokestatic coerce2Double(Ljava/lang/Object;)D
  dmul      // returns mul result, so always double
  dreturn
```




JavaScript has a lot of magic in its number coercion

```
var dict = Object.create(null);  
var key  = 'valueOf';  
  
//later  
dict[key] = formatHarddriveFunction;  
  
//much later  
dict++;
```

... and this turns into “10”, of course

`++ [[]][+[]]+[+[]]`

`===`

`“10”`



Brendan

Fibonacci calculator

```
function fib(_) {  
    for(_=[+[],++[[]][+[]],+[],_],_[++[++[++[[]][+[]]]  
        [+[]]][+[]]]=(((_[++[++[++[[]][+[]]][+[]]][+[]]]-  
        (++[[]][+[]]))&(((--[[]][+[]])>>>(++[[]][+[]]))))  
    ===(_[++[++[++[++[[]][+[]]][+[]]][+[]]][+[]]]-  
    (++[[]][+[]]))?(_[++[++[[]][+[]]][+[]]][+[]]]=  
    ++[[]][+[]],_[++[++[++[++[[]][+[]]][+[]]][+[]]][+[]]]-  
    (++[[]][+[]])):++;_[++[++[++[++[[]][+[]]][+[]]][+[]]]  
    [+[]]]--;_[++[]]=(_[++[[]][+[]]]+=  
    _[++[++[[]][+[]]][+[]]][+[]]]=_[++[]]+_[++[[]][+[]]])-  
    _[++[]]);  
    return _[++[++[++[[]][+[]]][+[]]][+[]]];  
}
```



Callsite specialization

- We can, and do, use static callsite types though.
- (ignore `int` overflows for a bit)

```
// Even if square is replaced, callsite type is not  
// It always takes a number, always returns a number  
var a = b * square(17.0);
```

Callsite specialization

- We can, and do, use static callsite types though.
- (ignore `int` overflows for a bit)

```
// Even if square is replaced, callsite type is not  
// It always takes a number, always returns a number  
var a = b * square(17.0);
```

```
square(D) D  
  dload 0  
  dup  
  dmul  
  dreturn
```

Callsite specialization

- We can, and do, use static callsite types though.
- (ignore `int` overflows for a bit)

```
// Even if square is replaced, callsite type is not
// It always takes a number, always returns a number
var a = b * square(17.0);
square = function(x) { return x + "string"; }
```

```
square(D) D
  dload 0
  dup
  dmul
  dreturn
```



Callsite specialization

```
square(Ljava/lang/Object;)Ljava/lang/Object;  
    aload 0  
    ldc "string"  
    JS_ADD(Ljava/lang/Object;Ljava/lang/Object);Ljava/lang/Object;  
    areturn
```



Callsite specialization

```
square (Ljava/lang/Object;) Ljava/lang/Object;  
    aload 0  
    ldc "string"  
    JS_ADD (Ljava/lang/Object; Ljava/lang/Object;) Ljava/lang/Object;  
    areturn
```

```
revert_square (D) D  
    dload 0  
    coerceToJSObject (D) Ljava/lang/Object; # param filter  
    invokedynamic square (Ljava/lang/Object;) Ljava/lang/Object;  
    coerceToDouble (Ljava/lang/Object;) D  
    dreturn
```


Callsite specialization

```
square (Ljava/lang/Object;) Ljava/lang/Object;  
    aload 0  
    ldc "string"  
    JS_ADD (Ljava/lang/Object; Ljava/lang/Object;) Ljava/lang/Object;  
    areturn
```

```
revert_square (D) D  
    dload 0  
    coerceToJSObject (D) Ljava/lang/Object; # param filter  
    invokeDynamic square (Ljava/lang/Object;) Ljava/lang/Object;  
    coerceToDouble (Ljava/lang/Object;) D  
    dreturn
```

NO EXPLICIT BYTECODE



Static compile time types bring us
performance,
[But they are too rare to take us all
the way]



Type Specialization

```
function am3(i,x,w,j,c,n) {  
  var this_array = this.array;  
  var w_array     = w.array;  
  
  var xl = x&0x3fff, xh = x>>14;  
  while(--n >= 0) {  
    var l = this_array[i]&0x3fff;  
    var h = this_array[i++]>>14;  
    var m = xh*l+h*xl;  
    l = xl*l+((m&0x3fff)<<14)+w_array[j]+c;  
    c = (l>>28)+(m>>14)+xh*h;  
    w_array[j++] = l&0xffffffff;  
  }  
  return c;  
}
```

Type Specialization – Prove ints

```
function am3(i,x,w,j,c,n) {  
  var this_array = this.array;  
  var w_array     = w.array;  
  
  var x1 = x&0x3fff, xh = x>>14;  
  while(--n >= 0) {  
    var l = this_array[i]&0x3fff;  
    var h = this_array[i++]>>14;  
    var m = xh*l+h*x1;  
    l = x1*l+ ( (m&0x3fff)<<14)+w_array[j]+c;  
    c = (l>>28)+(m>>14)+xh*h;  
    w_array[j++] = l&0xffffffff;  
  }  
  return c;  
}
```

Type Specialization – Prove doubles

```
function am3(i,x,w,j,c,n) {  
  var this_array = this.array;  
  var w_array    = w.array;  
  
  var xl = x&0x3fff, xh = x>>14;  
  while(--n >= 0) {  
    var l = this_array[i]&0x3fff;  
    var h = this_array[i++]>>14;  
    var m = xh*l+h*xl;  
    l = xl*l+((m&0x3fff)<<14)+w_array[j]+c;  
    c = (l>>28)+(m>>14)+xh*h;  
    w_array[j++] = l&0xffffffff;  
  }  
  return c;  
}
```

Static range analysis – fold doubles to ints

```
function am3(i,x,w,j,c,n) {
  var this_array = this.array;
  var w_array     = w.array;

  var x1 = x&0x3fff, xh = x>>14; // x1 = max 32 bits, xh: 18 bits
  while(--n >= 0) {
    var l = this_array[i]&0x3fff; // l max 12 bits
    var h = this_array[i++]>>14; // h max (32-14) = 18 bits
    var m = xh*l+h*x1;           // will never overflow
    l = x1*l+((m&0x3fff)<<14)+w_array[j]+c;
    c = (l>>28)+(m>>14)+xh*h;
    w_array[j++] = l&0xffffffff;
  }
  return c;
}
```

Static range analysis

```
function am3(i,x,w,j,c,n) {
  var this_array = this.array;
  var w_array     = w.array;

  var x1 = x&0x3fff, xh = x>>14; // x1 = max 32 bits, xh = 18 bits
  while(--i>=0) {
    var l = this_array[i]&0x3fff; // l max 12 bits
    var h = this_array[i++]>>14; // h max (32-14) = 18 bits
    var m = xh*l+h*x1; // will never overflow
    l = x1*l+((m&0x3fff)<<14)+w_array[j]+c;
    c = (l>>28)+(m>>14)+xh*h;
    w_array[j++] = l&0xffffffff;
  }
  return c;
}
```



Do we need our own inlining as well?



Do we need our own inlining as well?

We can statically prove a few primitive numbers from
callsites to `am3`.

Not from all of them.

Runtime callsite is really:

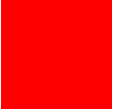
`(Ljava/lang/Object;IILjava/lang/Object;III)I`

Statically unprovable, though



Summary – Static analysis

- Just ignore all primitive types – use boxing everywhere and `axxx` instructions
 - Way too slow. The JVM is nowhere near being able to cope with that amount of boxing, and probably never will



Summary – Static analysis

- Just ignore all primitive types – use boxing everywhere and `axxx` instructions
 - Way too slow. The JVM is nowhere near being able to cope with that amount of boxing, and probably never will
- Use what primitives we can
 - Definitely gives us performance, depending on the amount of statically provable primitives



Summary – Static analysis

- Just ignore all primitive types – use boxing everywhere and `axxx` instructions
 - Way too slow. The JVM is nowhere near being able to cope with that amount of boxing, and probably never will
- Use what primitives we can
 - Definitely gives us performance, depending on the amount of statically provable primitives
- Add static range checking
 - Gives us another 30% or so



Summary – Static analysis

- Just ignore all primitive types – use boxing everywhere and `axxx` instructions
 - Way too slow. The JVM is nowhere near being able to cope with that amount of boxing, and probably never will
- Use what primitives we can
 - Definitely gives us performance, depending on the amount of statically provable primitives
- Add static range checking
 - Gives us another 30% or so
- Augment CFG with usedef chains to establish param types



But soon... static analysis won't get
us further unless we build our own
native JavaScript runtime



But soon... static analysis won't get
us further unless we build our own
native JavaScript runtime

Become adaptive/dynamic/optimistic



Statically provable callsites for am3

- . (Object, int, Object, Object, double, int, Object)Object
- . (Object, Object, Object, Object, double, int, int)Object
- . (Object, Object, double, Object, double, Object, double)Object
- . (Object, Object, Object, Object, double, int, int)Object
- . (Object, int, int, Object, double, int, Object)Object
- . (Object, int, Object, Object, Object, int, Object)Object



In fact they are...

- . (Object, int, int, Object, int, int, int)Object
- . (Object, int, int, Object, int, int, int)Object
- . (Object, int, int, Object, int, int, int)Object
- . (Object, int, int, Object, int, int, int)Object
- . (Object, int, int, Object, int, int, int)Object
- . (Object, int, int, Object, int, int, int)Object



In fact they are...

- . (Object, int, int, Object, int, int, int)Object
 - . (Object, int, int, Object, int, int, int)Object
 - . (Object, int, int, Object, int, int, int)Object
 - . (Object, int, int, Object, int, int, int)Object
 - . (Object, int, int, Object, int, int, int)Object
 - . (Object, int, int, Object, int, int, int)Object
-
- . We know this when linking at runtime



In fact they are...

- `(Object, int, int, Object, int, int, int)Object`
 - `(Object, int, int, Object, int, int, int)Object`
 - `(Object, int, int, Object, int, int, int)Object`
 - `(Object, int, int, Object, int, int, int)Object`
 - `(Object, int, int, Object, int, int, int)Object`
 - `(Object, int, int, Object, int, int, int)Object`
-
- We know this when linking at runtime
 - Use this signature to generate an optimistic version of `am3`, guard the types
 - Just because it's `int` right now, doesn't mean it's not undefined later. Guard required.



In fact they are...

- . `(Object, int, int, Object, int, int, int)Object`
 - . `(Object, int, int, Object, int, int, int)Object`
 - . `(Object, int, int, Object, int, int, int)Object`
 - . `(Object, int, int, Object, int, int, int)Object`
 - . `(Object, int, int, Object, int, int, int)Object`
 - . `(Object, int, int, Object, int, int, int)Object`
-
- . We know this when linking at runtime
 - . Use this signature to generate an optimistic version of `am3`, guard the types
 - . Just because it's `int` right now, doesn't mean it's not undefined later. Guard required.
 - . x2 Performance

We really want to use ints where we can

- `x++` pessimistic: `x` is double (if no static range analysis can prove otherwise)
- Having a double as a loop counter is slow
 - Loop unrolling doesn't work for non integer strides
 - Factor ~50 in improvement if replacing with ints

```
function f() {  
    var x = 0;  
    while (x < y) {  
        x++;  
    }  
    return x;  
}
```



We really want to use ints where we can

- All non-bitwise arithmetic can potentially overflow
- The + operator is the worst, as it can take any object
- Experiment: TypeScript frontend
 - A lot more performance with no further mods
 - Nashorn performs well with known primitive int types

```
function f() {  
    var x = 0;  
    while (x < y) {  
        x++; // dadd? iadd with overflow check?  
    }  
    return x;  
}
```



Using ints, problem 1 of 2 – Overflow check overhead

```
static int addExact(int x, int y) {  
    int result = x + y;  
    if ((x ^ result) & (y ^ result) < 0) {  
        throw new ArithmeticException("int overflow")  
    }  
    return result;  
}
```

```
function f() {  
    var x = 0;  
    while (x < y) {  
        x = addExact(x, 1);  
    }  
    return x;  
}
```

This is actually pretty much as slow as the `dadd` alone
Not sometimes, but often.



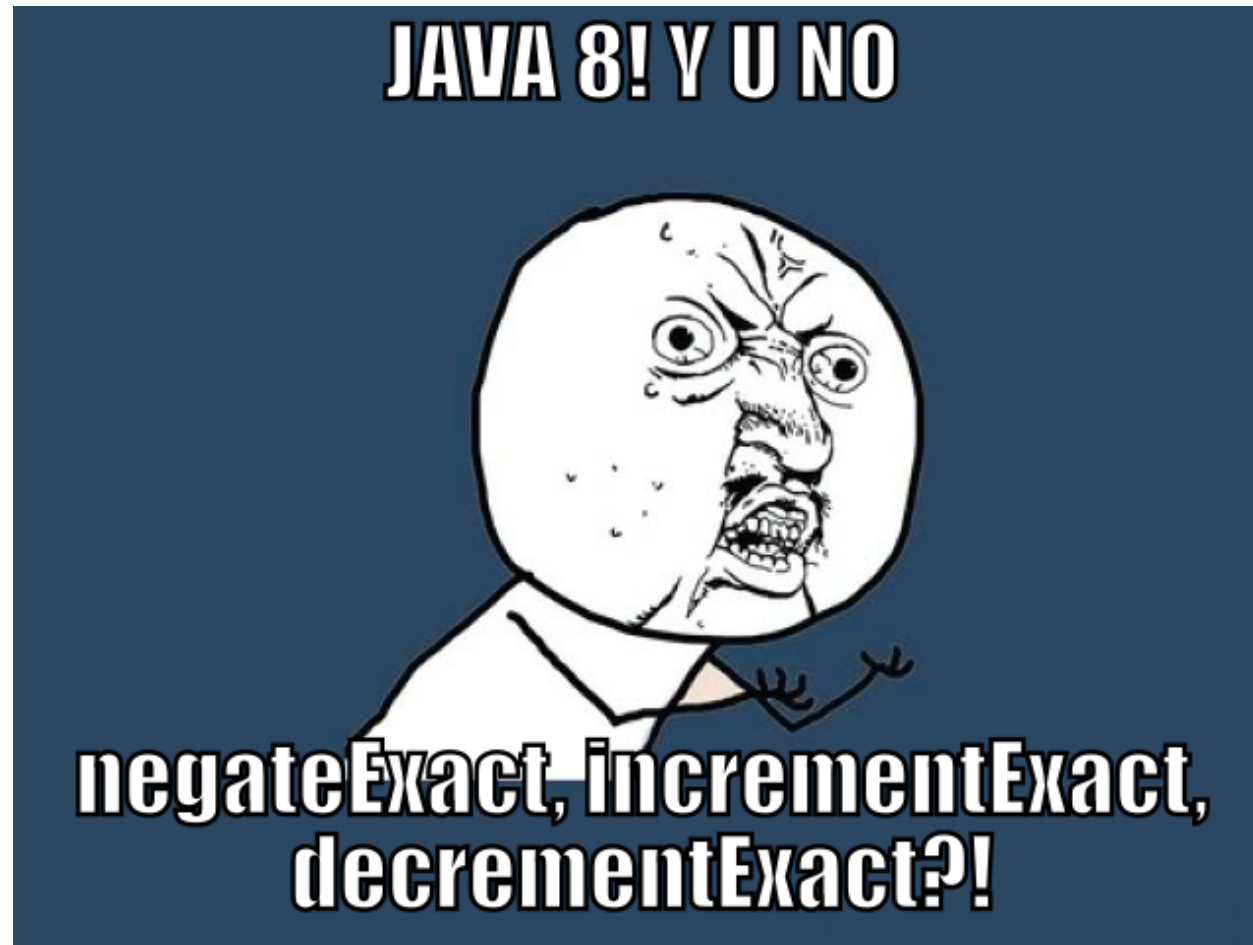
Solution: Intrinsicify math operations

- Java 8: `addExact`/`subExact`/`mulExact`
- Intrinsicify them
- Basically `addExact` is just

```
    add eax, edx
    jo fail
    ret
fail:
    //slow stuff
```

- < 10-15% slower than just the `iadd` when it doesn't fault
- Twice the speed of the non-intrinsicified version with `xors`
- Only slightly faster than `dadd`, but enables everything

Solution: Intrinsicify math operations





<pre>function f() { var x = 0; while (x < y) { x = addExact(x, 1); } return x; }</pre>	<pre>iconst_0 istore_0 while: iload_0 invokedynamic get_y()I if_icmpge exit iload_0 iconst_1 invokestatic addExact //intrinsic goto while exit: istore_0 ireturn</pre>
---	---

This is almost native-fast with add intrinsic and the int specialization.



<pre>function f() { var x = 0; while (x < y) { x = addExact(x, 1); } return x; }</pre>	<pre>iconst_0 istore_0 invokedynamic get y()I //check primitive istore_1 while: iload_0 iload_1 // y if_icmpge exit iload_0 iconst_1 invokestatic addExact //intrinsic goto while exit: istore_0 ireturn</pre>
---	---

(One more optimization: is y loop invariant? It may be a getter with side effects or anything as this is JavaScript hell... Hotspot won't be able to tell with the indy)



```
iconst_0
istore_0
invokedynamic get y()I //check primitive
istore_1
while:
iload_0
iload_1    // y
if_icmpge exit
iload_0
iconst_1
invokestatic addExact //intrinsic
goto while
exit:
istore_0
ireturn
```

Native-fast



We really want to use ints where we can

Very common instance of same problem.

```
function f() {  
    return 17 + array[3];  
}  
...  
bipush 17  
aload 2 //scope  
invokedynamic get:array(Ljava/lang/Object;)Ljava/lang/Object;  
aload 2  
iconst_3  
invokedynamic getElem(Ljava/lang/Object;I)Ljava/lang/Object;  
invokedynamic ADD:OIO_I(ILjava/lang/Object;)Ljava/lang/Object;  
areturn
```



We really want to use ints where we can

Very common instance of same problem.

```
function f() {  
    return 17 + array[3];  
}  
...  
bipush 17  
aload 2 //scope  
invokedynamic get:array(Ljava/lang/Object;)Ljava/lang/Object;  
aload 2  
iconst_3  
invokedynamic getElem(Ljava/lang/Object;I)I  
invokestatic Math.addExact  
ireturn
```



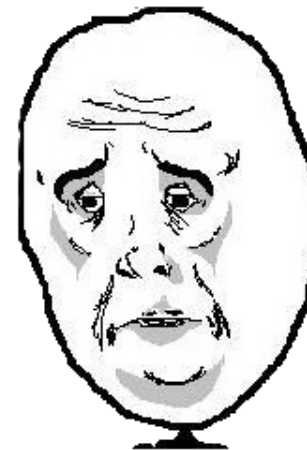
Using ints problem 2 of 2 – erroneous assumptions

- So what do we do if we overflow or miss an assumption?
- Bytecode is strongly typed, so we can't reuse the same code
- Throw errors or add guards/version code

Using ints problem 2 of 2 – erroneous assumptions

- So what do we do if we overflow or miss an assumption?
- Bytecode is strongly typed, so we can't reuse the same code
- ~~Throw errors or add guards/version code~~

```
if (x < y) {  
    x &= 1;  
    if (x < 2) {  
        x *= 2;  
        if (k) {  
            x += "string"  
            //keep branching  
        }  
    }  
}  
return x; //hope this is an int
```



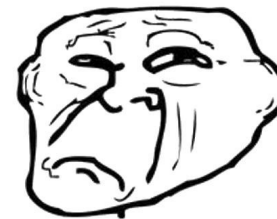


So add a catch block, take a continuation and jump to a less specialized version of the code



So add a catch block, take a **continuation** and jump to a less specialized version of the code

Uh-oh...





Continuations, you say?

Start out with

```
...  
ALOAD w_array  
ILOAD j  
  
INVOKEDYNAMIC dyn:getElem(I) I  
...  
IADD  
...
```



Continuations, you say?

Mark callsite optimistic, tag it with a program point

```
...  
ALOAD w_array  
ILOAD j  
  
INVOKEDYNAMIC dyn:getElem(I)I [optimistic | pp 17]  
...  
IADD  
...
```



Continuations, you say?

Add a return value filter throwing an Exception
if we return a non-int type

```
public class UnwarrantedOptimismException extends Exception {  
    ...  
    public int getProgramRestartPointId() { ... };  
    public Object getReturnedValue() { ... };  
}
```



Continuations, you say?

Send a message to the caller to regenerate the method

```
try {  
    ...  
    ALOAD w_array  
    ILOAD j  
    // make sure bc stack is written to locals  
    INVOKEDYNAMIC dyn:getElem(I)I [optimistic | pp 17]  
    ...  
    IADD  
    ...  
} catch (UnwarrantedOptimismException e) {  
    // ask linker to regenerate method  
    throw new RewriteException(e.getId(), e.getReturnValue(), locals);  
}
```

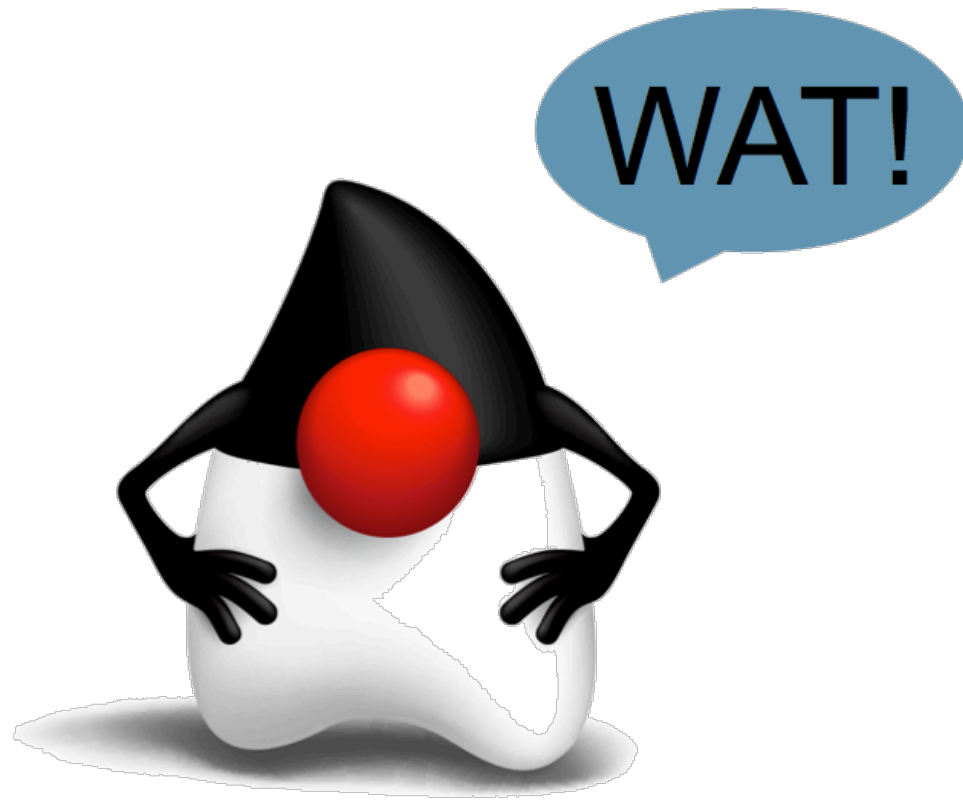


Continuations, you say?

- We know when we are relinking a rewritable method
- Add a `MethodHandles.catchException` for `RewriteException`
- Catch triggers recompilation, with the failed callsite made more pessimistic.
- Also generates and invokes a “rest of” method

```
restOfMethod(RewriteException e) {  
    // store to locals e.getLocals();  
    // ...  
    // all code after invokedynamic that failed with  
    // maximum pessimism  
    // (can never throw UnwarrantedOptimismException)  
    return pessimisticReturnValue;  
}
```

The JVM situation





JVM issues

- Java 7
 - Pretty quickly started giving us the infamous `NoClassDefFoundError` bug
 - Circumvented by running with everything in `bootclasspath` (Eww...)
- Java 8
 - A lot of C++ was reimplemented as LambdaForms
 - Initially, 10% of Java 7 performance. ☹



```
print(Math.round(0.5));
```

WTF?

```
▼ New_configuration [Java Application]
  ▼ jdk.nashorn.tools.Shell at localhost:54014
    ▼ Thread [main] (Suspended (breakpoint at line 618 in NativeMath))
      NativeMath.round(Object, Object) line: 618
      LambdaForm$DMH.invokeStaticInit_LL_L(Object, Object, Object) line: not available
      LambdaForm$DMH.invokeSpecial_LLL_L(Object, Object, Object, Object) line: not available
      LambdaForm$NamedFunction.invoke_LLL_L(MethodHandle, Object[]) line: 1102
      LambdaForm$DMH.invokeStatic_LL_L(Object, Object, Object) line: not available
      LambdaForm$NamedFunction.invokeWithArguments(Object...) line: 1136
      LambdaForm.interpretName(LambdaForm$Name, Object[]) line: 625
      LambdaForm.interpretWithArguments(Object...) line: 604
      LambdaForm$LFIL.interpret_L(MethodHandle, Object, Object, Object) line: not available
      LambdaForm$NamedFunction.invoke_LLL_L(MethodHandle, Object[]) line: 1102
      LambdaForm$DMH.invokeStatic_LL_L(Object, Object, Object) line: not available
      LambdaForm$NamedFunction.invokeWithArguments(Object...) line: 1136
      LambdaForm.interpretName(LambdaForm$Name, Object[]) line: 625
      LambdaForm.interpretWithArguments(Object...) line: 604
      LambdaForm$LFIL.interpret_L(MethodHandle, Object, Object, double) line: not available
      LambdaForm$DMH.invokeSpecial_LLLD_L(Object, Object, Object, Object, double) line: not available
      LambdaForm$NFI.invoke_LLLD_L(MethodHandle, Object[]) line: not available
      LambdaForm$DMH.invokeStatic_LL_L(Object, Object, Object) line: not available
      LambdaForm$NamedFunction.invokeWithArguments(Object...) line: 1136
      LambdaForm.interpretName(LambdaForm$Name, Object[]) line: 625
      LambdaForm.interpretWithArguments(Object...) line: 604
      LambdaForm$LFIL.interpret_L(MethodHandle, Object, Object, Object, double) line: not available
      LambdaForm$NFI.invoke_LLLD_L(MethodHandle, Object[]) line: not available
      LambdaForm$DMH.invokeStatic_LL_L(Object, Object, Object) line: not available
      LambdaForm$NamedFunction.invokeWithArguments(Object...) line: 1136
      LambdaForm.interpretName(LambdaForm$Name, Object[]) line: 625
      LambdaForm.interpretWithArguments(Object...) line: 604
      LambdaForm$LFIL.interpret_L(MethodHandle, Object, Object, double) line: not available
      LambdaForm$MH.linkToCallSite(Object, Object, double, Object) line: not available
      Script$b.runScript(ScriptFunction, Object) line: 1
      LambdaForm$DMH.invokeStatic_LL_L(Object, Object, Object) line: not available
      LambdaForm$MH.invokeExact_MT(Object, Object, Object, Object) line: not available
      FinalScriptFunctionData(ScriptFunctionData).invoke(ScriptFunction, Object, Object...) line: 512
      ScriptFunctionImpl(ScriptFunction).invoke(Object, Object...) line: 202
      ScriptRuntime.apply(ScriptFunction, Object, Object...) line: 346
      Shell.apply(ScriptFunction, Object) line: 385
      Shell.runScripts(Context, ScriptObject, List<String>) line: 314
      Shell.run(InputStream, OutputStream, OutputStream, String[]) line: 178
      Shell.main(InputStream, OutputStream, OutputStream, String[]) line: 142
      Shell.main(String[]) line: 121
```

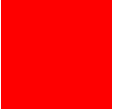
[illegible]

401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000

[illegible]

JVM issues

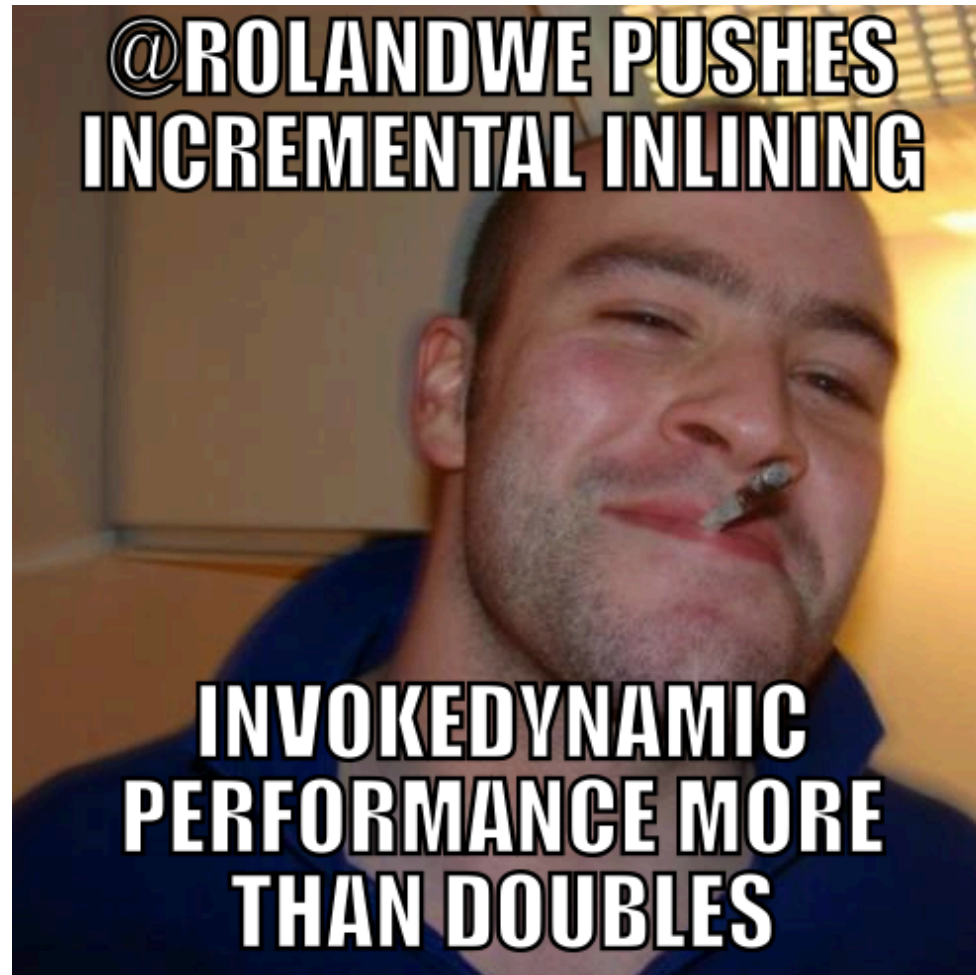




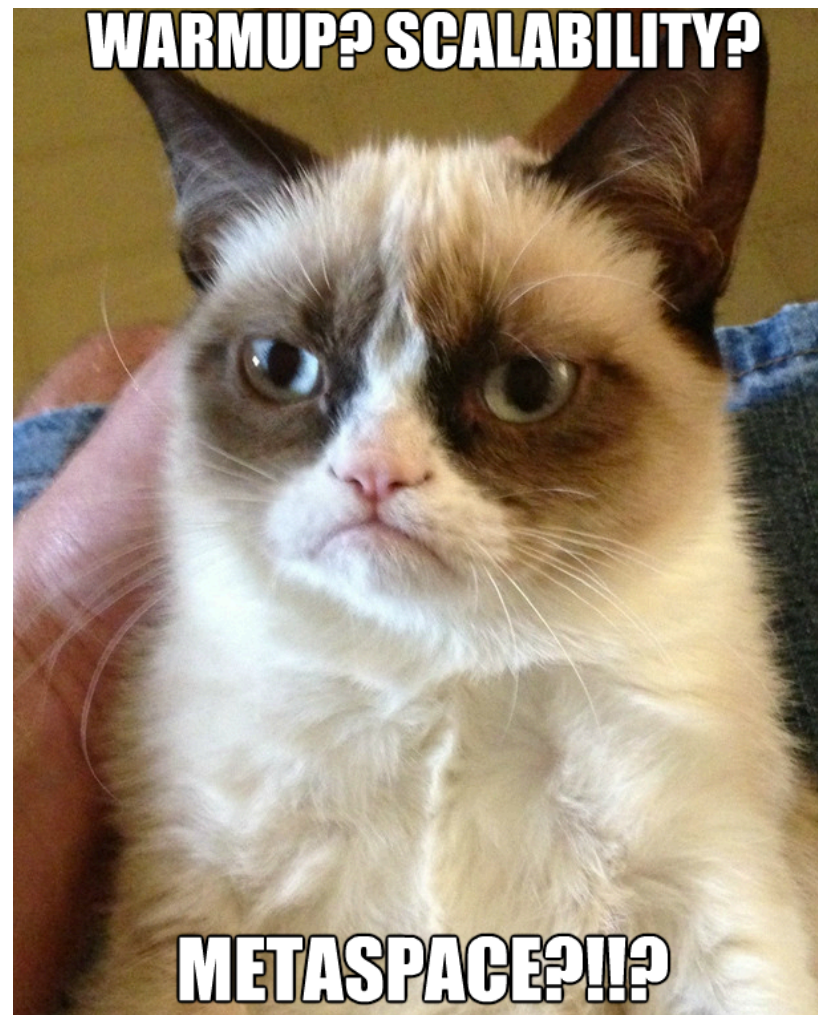
JVM issues

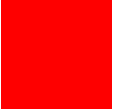
- Many inlining problems
 - Even, traditionally, for normal Java code – add a code line, 50% of performance disappears
 - Seen that from time to time with HotSpot
 - Relevant in our quick paths in Nashorn too
- LambdaForms & MethodHandles
 - Tremendous pressure on inlining, lambda form classes also on metaspace
- Discovered a few very old bugs in C2 inliner
 - E.g: dead nodes counted as size.

JVM issues



JVM issues





JVM issues

- LambdaForms compile a lot of code, generate a lot of metaspace stress
- If we have to have LambdaForms, they might not be able to remain in bytecode land?
- Inlining, despite tweaking has a lot of problems that remain to be solved
- Boxing removal boxing removal boxing removal
 - (probably enabled by local escape analysis)

JVM issues

- MethodHandle.invoke (not exact) is slow

```
public class Test {  
    private final static MethodHandle CALC =  
        MethodHandles.publicLookup().findStatic(  
            Test.class, "calc", int.class, int.class, Object.class);  
  
    static int test() throws Throwable {  
        MethodHandle mh = CALC;  
        Object aString = "A";  
        int a = mh.invoke(1, aString);  
        int b = mh.invoke(2, "B");  
        Integer c = mh.invoke((Integer)3, 3);  
        return a+b+c;  
    }  
  
    static int calc(int x, Object o) {  
        return x + o.hashCode();  
    }  
}
```



JVM issues

- MethodHandle.invoke (not exact) is slow

```
public class Test {  
    private final static MethodHandle CALC =  
        MethodHandles.publicLookup().findStatic(  
            Test.class, "calc", int.class, int.class, Object.class);  
  
    static int test() throws Throwable {  
        return 140;  
    }  
  
    static int calc(int x, Object o) {  
        return x + o.hashCode();  
    }  
}
```



JVM issues

- Still artifacts here. We do ugly stuff in Java like

```
@Override
public long getLong(final long key) {
    final int index = ArrayIndex.getArrayIndex(key);
    final ArrayData array = getArray();

    if (array.has(index)) {
        return array.getLong(index);
    }

    return getLong(index, convertKey(key));
}
```



JVM issues

- Still artifacts here. We do ugly stuff in Java like

```
@Override
public long getLong(final double key) {
    final int index = ArrayIndex.getArrayIndex(key);
    final ArrayData array = getArray();

    if (array.has(index)) {
        return array.getLong(index);
    }

    return getLong(index, convertKey(key));
}
```



JVM issues

- Still artifacts here. We do ugly stuff in Java like

```
@Override
public long getLong(final Object key) {
    final int index = ArrayIndex.getArrayIndex(key);
    final ArrayData array = getArray();

    if (array.has(index)) {
        return array.getLong(index);
    }

    return getLong(index, convertKey(key));
}
```



JVM issues

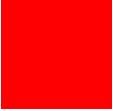
- Still artifacts here. We do ugly stuff in Java like

```
@Override
public long getLong(final int key) {

    final ArrayData array = getArray();

    if (array.has(key)) {
        return array.getLong(key);
    }

    return getLong(key, convertKey(key));
}
```



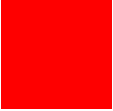
War story: warmup

- Indy intrinsically needs bootstrapping
- Every call site contributes to warmup
- LambdaForms contribute to warmup
- Tiered compilation has gone back and forth.
 - Peak performance is reached sooner, even without C2 compiling all the methods
 - Added deviation has been very large
 - C2 is slow



Another war story: Metaspace

- Runtime didn't know about anonymous classes
- Build b58-b74 were broken ☹️
- Compressed klass pointers gave us a fixed size 100 MB default klass pointer chunk ☹️
- Metaspace allocated from metaspace pool subject to fragmentation. Chunks went 5% full to different classloaders
- HotSpot did not hand back deallocated Metaspace memory to the OS



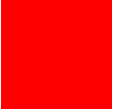
Future work – Nashorn

- Optimistic code everywhere
- Static analysis/IR
- Field representations
 - Objects only, dual fields, `sun.misc.TaggedArray`
(`TaggedObject?`)
- Parallelism



Future work - JVM

- Boxing removal (probably requires Local EA)
- `sun.misc.TaggedArray`?
- Intrinsicify `Math.addExact` and friends
 - Done!
- `MethodHandle.invoke` must be fast
- LambdaForms
 - Caching for footprint?
 - Replacing LambdaForms with something else?
 - Get them out of class/bytecode land



Future work - JVM

- Is bytecode even the correct format to do this entire in
 - Pluggable frontends?
 - More magic: I probably really need to talk to my compiler
 - Or have my compiler talk to me

Nashorn current performance status

- (Very) initial POC after 2.5 weeks of work:
 - Broke out `octane.crypto.am3` – the hotspot in the Crypto benchmark in octane.
 - Turned it into microbenchmark

```
function am3(i, x, w, j, c, n) {
  var this_array = this.array;
  var w_array    = w.array;
  var xl = x & 0x3fff, xh = x >> 14;

  while(--n >= 0) {
    var l = this_array[i] & 0x3fff;
    var h = this_array[i++] >> 14;
    var m = xh * l + h * xl;
    l = xl * l + ((m & 0x3fff) << 14) + w_array[j] + c;
    c = (l >> 28) + (m >> 14) + xh * h;
    w_array[j++] = l & 0xffffffff;
  }

  return c;
}
```



Nashorn current performance status

- Runtime
 - Rhino (with `-opt 9`): 34.6 s
 - Nashorn tip: 10.8 s
 - V8 1.3 s



Nashorn with optimistic types

- Runtime
 - Rhino (with `–opt 9`): 34.6 s
 - Nashorn tip: 5.8 s
 - V8 1.3 s



Add JVM math intrinsics...

- Runtime
 - Rhino (with `-opt 9`): 34.6 s
 - Nashorn tip: 4.4 s
 - V8 1.3 s



Patch JVM to keep more type info while inlining...

- Runtime
 - Rhino (with `-opt 9`): 34.6 s
 - Nashorn tip: 2.5 s
 - V8 1.3 s

Talk to us

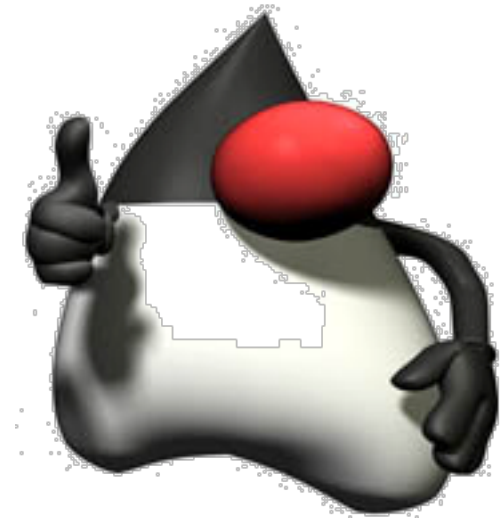
- Tweet us: @lagergren, @wickund, @asz, @hannesw, @sundararajan_a
- <http://blogs.oracle.com/nashorn>
- nashorn-dev@openjdk.java.net
- mlvm-dev@openjdk.java.net





Thank you!

Q&A?



@lagergren

ORACLE®