Great Thundering Rhinos!
(an expedition into JavaScript optimization)

John Rose
Oracle Corporation
9/21/2010
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 decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Overview

• JavaScript presents difficult implementation obstacles
• JDK 7 provides an excellent toolkit for implementors
• Case study: Optimizing Rhino with invokedynamic
What’s hard about JavaScript

• A highly dynamic language
  – name bindings are subject to change (no “final”)
  – even methods are mutable name bindings
  – no explicit types; all typing is latent and dynamic

• Objects work like hashmaps
  – any statement can add a field to any object
  – arrays have indexed fields but can be given named fields
  – inheritance is subject to change (set the “prototype”)

• Many historical irregularities and intrinsic methods
What’s hard about JavaScript

Scheduler.prototype.schedule = function () {
    this.currentTcb = this.list;
    while (this.currentTcb != null) {
        if (this.currentTcb.isHeldOrSuspended()) {
            this.currentTcb = this.currentTcb.link;
        } else {
            this.currentId = this.currentTcb.id;
            this.currentTcb = this.currentTcb.run();
        }
    }
};
What’s not so hard about JavaScript

• Designed from known and traditional elements
  – objects similar to Smalltalk or Self
  – closures similar to Smalltalk or Lisp
  – basic types not too different from Java (& 700 others)

• There is a mature literature on optimizing such things
  – Chambers, Ungar, Lee “An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes” OOPSLA 1989
JavaScript is robustly popular, and growing

Monthly Commits (Percent of Total)
The lines show the count of monthly commits made by source code developers. Commits including multiple languages are counted once for each language. More

• http://www.ohloh.net/languages/compare
Let’s get the JVM involved

- JVMs provide excellent tools for implementors
- invokedynamic (JDK 7) adds some missing items
- JVM optimizations designed for Java help JavaScript
compiler tactics
  delayed compilation
  Tiered compilation
  on-stack replacement
  delayed reoptimization
  program dependence graph representation
  static single assignment representation
proof-based techniques
  exact type inference
  memory value inference
  memory value tracking
  constant folding
  reassociation
  operator strength reduction
  null check elimination
  type test strength reduction
  type test elimination
  algebraic simplification
  common subexpression elimination
flow-sensitive rewrites
  conditional constant propagation
  dominating test detection
  flow-carried type narrowing
  dead code elimination
language-specific techniques
  class hierarchy analysis
  devirtualization
  symbolic constant propagation
  autobox elimination
  escape analysis
  lock elision
  lock fusion
  de-reflection
  speculative (profile-based) techniques
  optimistic nullness assertions
  optimistic type assertions
  optimistic type strengthening
  optimistic array length strengthening
  untaken branch pruning
  optimistic N-morphic inlining
  branch frequency prediction
  call frequency prediction
  memory and placement transformation
  expression hoisting
  expression sinking
  redundant store elimination
  adjacent store fusion
  card-mark elimination
  merge-point splitting
loop transformations
  loop unrolling
  loop peeling
  safepoint elimination
  iteration range splitting
  range check elimination
  loop vectorization
  global code shaping
  inlining (graph integration)
  global code motion
  heat-based code layout
  switch balancing
  throw inlining
  control flow graph transformation
  local code scheduling
  local code bundling
  delay slot filling
  graph-coloring register allocation
  linear scan register allocation
  live range splitting
  copy coalescing
  constant splitting
  copy removal
  address mode matching
  instruction peepholing
  DFA-based code generator
Let’s get the JVM involved!

- JVMs provide excellent tools for implementors
- invokedynamic (JDK 7) adds some missing items
- JVM optimizations designed for Java help JavaScript
Reality is a simulation

Java language fictions

Java VM features

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

Checked exceptions
Generics
Enums
Overloading
Constructor chaining
Program analysis

JavaScript language fictions

Object properties
Dynamic typing
'eval'
Closures
Prototypes
Regular expressions

Primitive types+ops
Object model
Memory model
Dynamic linking
Access control
GC
Threads
Rhino redux

• Rhino JavaScript engine, Netscape ca. 1997
  – one of the first languages built on the JVM
  – created in the same organization as the original engine
• And, it never went away.
  – Now in 1.7R2 release (ECMA 262, 357)
  – http://www.mozilla.org/rhino
  – Open source; bundled in JDK
Benefits of synergy

• JavaScript: the right tool for some jobs
  – as opposed to the whole environment

• Easy call-outs from JavaScript to Java APIs
  – and vice-versa
  – NxN integration with other languages via a MOP

• Better access to JVM strengths
  – threads, security, scaling
  – code generation, optimization, management
  – deployment options
Rhino performance

- Originally competitive with corresponding C engine
  - Now rather slow
  - Newer engines much faster: V8, Nitro, JaegerMonkey
- Includes bytecode compilation mode
  - This open-codes an AST interpreter dispatch loop
  - Typical performance gains for such things $\approx 2x$
- Performance $\approx 20x$ slower than current leaders
Rhino performance diagnosis

• Object bloat
  – Objects are maps, with “lots of little gears”
  – Even integers are 2-4 words

• No leverage from JVM classes
  – Mutability prevents direct use of JVM classes
  – Most JavaScript objects “look the same” to the JVM
  – JavaScript object layout and behavior is mysterious

• Too many indirections
  – Cause memory stalls
  – Prohibit inlining and other static analysis
Rhino performance prescription

• Linearize the objects
  – Mutable system of layouts layered on top of JVM classes
  – Each layout maps symbolic property names to slot offsets
  – An object can change layout if it adds or deletes properties

• Profile by object layouts
  – Each call site profiles object layout (as well as JVM class)
  – Note: JVM-level call sites for all property use (get/set/call)
  – The JVM doesn’t know about layouts, but the runtime does

• Weave direct code for expected layouts
  – Done independently at each call site (profile point)
  – Method handles can be inlined into each other

• Work with JVM engineers as bottlenecks appear
A proof of concept project

- ≈1 month of prototyping, 12/2009
- Code sketches are posted on Kenai.com

davincimonkey
Example: Property access

```javascript
Scheduler.prototype.schedule = function () {
    this.currentTcb = this.list;
    while (this.currentTcb != null) {
        if (this.currentTcb.isHeldOrSuspended()) {
            this.currentTcb = this.currentTcb.link;
        } else {
            this.currentId = this.currentTcb.id;
            this.currentTcb = this.currentTcb.run();
        }
    }
};
```
Example: Property access

```java
void _c9_schedule(Scriptable js_this) throws Throwable {
    InvokeDynamic.#"set:currentTcb"(js_this,
        InvokeDynamic.list(js_this));
    for (;;) {
        Object object = InvokeDynamic.currentTcb(js_this);
        if (object == null || object == Undefined.instance) break;
        if ((boolean) InvokeDynamic.#"call:isHeldOrSuspended"(
            InvokeDynamic.currentTcb(js_this)) ) {
            InvokeDynamic.#"set:currentTcb"(js_this,
                InvokeDynamic.link(InvokeDynamic.currentTcb(js_this)));
        } else {
            InvokeDynamic.#"set:currentId"(js_this,
                InvokeDynamic.id(InvokeDynamic.currentTcb(js_this)));
            InvokeDynamic.#"set:currentTcb"(js_this,
                InvokeDynamic.#"call:run"(
                    InvokeDynamic.currentTcb(js_this)));
        }
    }
}
```
Example: Property access

```java
void _c9_schedule(Scriptable js_this) throws Throwable {
    InvokeDynamic.#"set:currentTcb"(js_this,
        InvokeDynamic.list(js_this));
    for (; ;) {
        Object object = InvokeDynamic.currentTcb(js_this);
        if (object == null || object == Undefined.instance)  break;
        if ((boolean) InvokeDynamic.#"call:isHeldOrSuspended"(
            InvokeDynamic.currentTcb(js_this)) ) {
            InvokeDynamic.#"set:currentTcb"(js_this,
                InvokeDynamic.link(InvokeDynamic.currentTcb(js_this)));
        } else {
            InvokeDynamic.#"set:currentId"(js_this,
                InvokeDynamic.id(InvokeDynamic.currentTcb(js_this)));
            InvokeDynamic.#"set:currentTcb"(js_this,
                InvokeDynamic.#"call:run"(
                    InvokeDynamic.currentTcb(js_this)));
        }
    }
}
```
Example: Property access

```java
object = InvokeDynamic.currentTcb(js_this)
```
JVM call architecture (before invokedynamic)
JVM call architecture (after invokedynamic)
Example: Property access

// object = InvokeDynamic.currentTcb(js_this)
// after Bootstrap Method creates call site:
static ClassyCallSite cs2199 = new ...;
    cs2199.profileReceiver(js_this);
    object = cs2199.slowGet(js_this);
Example: Property access

// object = InvokeDynamic.currentTcb(js_this)
// after ClassyCallSite decides it like a layout:
static final ClassyCallSite cs2199 = ...;
static final ClassyLayout lo2199
    = cs2199.receiverLayout();
static final MethodHandle st2199
    = lo2199.findSlot("currentTcb").getter();

if ((js_this instanceof HasLayout) &&
    ((HasLayout)js_this).getLayout() == lo2199) {
    object = sT2199.invokeExact(js_this);
} else {
    object = cs2199.slowGet(js_this);
}
Example: Property access

// object = InvokeDynamic.currentTcb(js_this)
//   pseudo-code

// test 1
if (js_this.getClass() != Struct4.class) goto slow;

// test 2
if (cs2199.getTarget() != expected2199)  goto slow;

// test 3
Struct4 s4_this = (Struct4) js_this;
if (s4_this.layout != lo2199)  goto slow;

object = s4_this.f3;  // slot #3
Example: Property access

• Test 1: Check for invokedynamic re-linking
  – This is “pull notification”, which is destined to go away
• Test 2: Check the JVM type of the reference
  – Normal result of JVM-level type profiling
• Test 3: Check the “ClassyLayout” of the reference
  – JVM doesn’t know about this, but it knows how to optimize
    – “ClassyCallSite” profiling manages layout prediction
• End result: Load a JVM object field (Struct4.s3)
  – This is as efficient as any Java field access.
Edge cases

• Test 1: The dynamic call site might relink itself.
• Test 2: A String or Integer might suddenly show up.
• Test 3: An unexpected ClassyLayout might show up.
• Load: The original object might have been mis-sized.
  – Start with a “Struct4” and add 100 properties.
  – Oops, should have started with bigger block of memory
• Every Classy object has an extension.
  – Struct4 has 1 layout, 4 fields, 1 extension.
  – The extension is a (hidden) resizable Object array.
  – Therefore, can have up to 2 billion slots.
Key JVM tricks

- Inlining (the über-optimization)
- Type profiling
- Editable call sites (invokedynamic)
  - Enable inline caches written by the user
- Low-level method handles for method and slot access
  - No need for reflection
- Path trimming (the “goto” in the pseudocode)
  - Dead paths are covered by “deoptimization”
Key language implementation tricks

- Object typestate (aka. layout, map, hidden class)
  - Profile it and trust the profile
  - Make state-specific fast paths

- Balanced mix of bytecode and method handles
  - Also AST, but that’s not in Rhino right now

- Inline caching (because each call site is different!)
  - This is enabled by invokedynamic

- Use native JVM types (Integer, String)
  - No language-specific wrapper classes

- Suppress seldom-used “context” arguments
Results

• Much better machine code for typical JavaScript
• Throughput increased about 4x, to within 50% of V8
• Consistent with similar studies
  – JRuby retargeting to invokedynamic (monomorphic caches)
  – PHP.reboot project (trace-based bytecodes after type profile)
  – Experiments with generic arithmetic (type profiling)
More work to do in the Rhino runtime...

- This Proof of Concept was done for *one* benchmark
  - Richards (an oldy-but-goody, taken from the V8 suite)
- There was lots of “slack” in the runtime support code
  - We removed some of it, but there’s lots more
- The bytecode generator was...

  Emacs

- Trace-based bytecode generation is promising.
  - Rémi Forax’s PHP.reboot project shows how to do this
More work to do in the JVM...

• Many more iterations of ad hoc JVM tuning
• Push-based invalidation for invokedynamic
• Cheaper ints: “fixnums”
  – Boxing is main bottleneck for dynamically-typed arithmetic
• Cheaper typestate: “species” (aka. split classes)
  – Will fold up Test 1 (class) and Test 3 (layout)
  – Will make objects smaller by 1 word
• Faster startup (saved images)
  – Investigation continues... This is a hard one.
“Fixnums” – tagged immediate pseudo-pointers

• In Java, primitives can be “autoboxed”
  – This convenience was added in JDK 5
• Boxing is expensive and tricky to optimize
  – In general it requires building a whole “wrapper” object
• Some older systems (Lisp, Smalltalk) are smarter
  – They use the object pointer itself to store the primitive value
  – The pointer is “tagged” to distinguish it from a real address
A list of integer values (before fixnums)

```
Arrays.asList(0x5551212, 0x411, 0x311)
```
A list of integer values (after fixnums)

Arrays.asList(0x5551212, 0x411, 0x311)

<table>
<thead>
<tr>
<th>Class:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
</tr>
<tr>
<td>0x5551212F</td>
</tr>
<tr>
<td>0x0000411F</td>
</tr>
<tr>
<td>0x0000311F</td>
</tr>
</tbody>
</table>

Memory is untouched by integers that fit into 28 bits
To find out more...

- Initial code sketches
  http://kenai.com/projects/davincimmonkey

- “Optimizing Invokedynamic”, Thalinger PPPJ 2010
  http://blogs.sun.com/jrose/entry/an_experiment_with_generic_arithmetic

- JVM Language Summit 2010
  http://wiki.jvmlangsummit.com

- Da Vinci Machine Project
  http://openjdk.java.net/projects/mlvm/

- “PHP Reboot” project
  http://wiki.jvmlangsummit.com/
  PHP.reboot:_a_post_JSR292_dynamic_language