Know Your Engines
How to Make Your JavaScript Fast

Dave Mandelin
June 15, 2011
O’Reilly Velocity
JavaScript now runs 10–100x faster than 5 years ago, fast on all major browsers.
Developers using it for new apps: interactive movies, games, photo editing, slides
I’m going to explain how it works to help you get the most out of these engines.
function f() {
    var sum = 0;
    for (var i = 0; i < N; ++i) {
        sum += i;
    }
}

function f() {
    eval('"");
    var sum = 0;
    for (var i = 0; i < N; ++i) {
        sum += i;
    }
}
Making JavaScript Fast

Or, Not Making JavaScript Slow

♦ How JITs make JavaScript not slow
♦ How not to ruin animation with pauses
♦ How to write JavaScript that’s not slow
The 2006 JavaScript Engine
Inside the 2006 JS Engine

// JavaScript source
e.innerHTML = n + " items";

// bytecode (AST in some engines)
tmp_0 = add var_1 str_3
setprop var_0 'innerHTML' tmp_0

Front End → Interpreter

Set innerHTML
DOM
Standard Library
Garbage Collector
Run the bytecode
Reclaim memory
Why it’s hard to make JS fast

Because

JavaScript is an untyped language.

untyped = no type declarations
Operations in an untyped language

\[ x = y + z \text{ can mean many things} \]

- if \( y \) and \( z \) are numbers, numeric addition
- if \( y \) and \( z \) are strings, concatenation
- and many other cases; \( y \) and \( z \) can have different types
Engine-Internal Types

JS engines use finer-grained types internally.

<table>
<thead>
<tr>
<th>JavaScript type</th>
<th>Engine type</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>32-bit* integer</td>
</tr>
<tr>
<td></td>
<td>64-bit floating-point</td>
</tr>
<tr>
<td>object</td>
<td></td>
</tr>
</tbody>
</table>

Different shapes:

{ a: 1 }
{ a: 1, b: 2 }
{ a: get ... }
{ a: 1, __proto__ = new C }
Values in an untyped language

Because JavaScript is untyped, the interpreter needs **boxed** values.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Boxed</th>
<th>Unboxed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>Storage</td>
<td>Computation</td>
</tr>
<tr>
<td>Examples</td>
<td>(INT, 55)</td>
<td>55</td>
</tr>
<tr>
<td>(STRING, “foo”)</td>
<td>“foo”</td>
<td></td>
</tr>
<tr>
<td>Definition</td>
<td>(type tag, C++ value)</td>
<td>C++ value</td>
</tr>
</tbody>
</table>

**only** boxed values can be stored in variables, **only** unboxed values can be computed with (+, *, etc)
Running Code in the Interpreter

Here’s what the interpreter must do to execute $x = y + z$:

- read the operation $x = y + z$ from memory
- read the boxed inputs $y$ and $z$ from memory
- check the types of $y$ and $z$ and choose the action
- unbox $y$ and $z$
- **execute the action**
- box the output $x$
- write the boxed output $x$ to memory

This is the only real work! Everything else is overhead.
The 2011 JavaScript Engine
Inside the 2011 JS Engine

THE SLOW ZONE

Front End

Interpreter

JIT Compiler

Type-Specializing JIT Compiler

Fast!

Compile to x86/x64/ARM

Ultra Fast!

JavaScript source

bytecode/AST

DOM

Garbage Collector

Standard Library

CPU

x86/x64/ARM

Fast!

mozilla
Running Code with the JIT

The basic JIT compiler on $x = y + z$:

- read the operation $x = y + z$ from memory
- read the inputs $y$ and $z$ from memory
- check the types of $y$ and $z$ and choose the action
- unbox $y$ and $z$
- **execute the action**
- box the output $x$
- write the output $x$ to memory

**CPU does it for us!**

**JIT code can keep things in registers**

**All Major Browsers**
Choosing the action in the JIT

- Many cases for operators like +
- Engines generate fast JIT code for “common cases”
  - number + number
  - string + string
- “Rare cases” run in the slow zone
  - number + undefined
JITs for Regular Expressions

• There is a separate JIT for regular expressions
• Regular expressions are generally faster than manual search
• Still in the slow zone:
  • Some complex regexes (example: backreferences)
  • Building result arrays (test much faster than exec)
Object Properties

```javascript
function f(obj) {
    return obj.a + 1;
}
```

- Need to search `obj` for a property named `a` **slow**
- May need to search prototype chain up several levels **super-slow**
- Finally, once we've found it, get the property value **fast!**
ICs: a mini-JIT for objects

• Properties become fast with **inline caching** (we prefer IC)
• Basic plan:
  1. First time around, search for the property in the Slow Zone
  2. But record the steps done to actually get the property
  3. Then JIT a little piece of code that does just that
Example Code:

```javascript
var obj1 = { a: 1, b: 2, c: 3 };  // shape=12, in position 1
var obj2 = { b: 2 };             // shape=15, in position 0

function f(obj) {
    return obj.b + 1;
}
```

Generated JIT Code:

```javascript
... 
jump slowPropAccess
continue_1: 
... 
```

```javascript
icStub_1:
    compare obj.shape, 12  
    jumpIfFalse slowPropAccess 
    load obj.props[1]  
    jump continue_1

icStub_2:
    compare obj.shape, 15  
    jumpIfFalse slowPropAccess 
    load obj.props[0]  
    jump continue_1

slowPropAccess:
    ... set up call 
    call ICGetProp ; C++ Slow Zone 
    jump continue_1
```
These are fast because of ICs

**Global Variable Access**

```javascript
var q = 4;
var r;

function f(obj) {
    r = q;
}
```

**Direct Property Access**

```javascript
var obj1 = { a: 1, b: 2, c: 3 }; var obj2 = { b: 2 }; 

function f(obj) {
    obj2.b = obj1.c;
}
```

**Closure Variable Access**

```javascript
var f = function() {
    var x = 1;
    var g = function() {
        var sum = 0;
        for (var i = 0; i < N; ++i) {
            sum += x;
        }
        return sum;
    }
    return g();
}
```
Prototypes don’t hurt much

```javascript
function A(x) {
  this.x = x;
}

function B(y) {
  this.y = y;
}
B.prototype = new A;

function C(z) {
  this.z = z;
}
C.prototype = new B;
```

Shape of new C objects determines prototype

-> IC can generate code that checks shape, then reads directly from prototype without walking
Many Shapes Slow Down ICs

What happens if many shapes of obj are passed to f?

```
function f(obj) {
    return obj.p;
}
```

ICs end up looking like this:

- `jumpIf shape != 12`
- `read for shape 12`
- `jumpIf shape != 15`
- `read for shape 15`
- `jumpIf shape != 6`
- `read for shape 6`
- `jumpIf shape != 16`
- `read for shape 16`
- `jumpIf shape != 22`
- `read for shape 22`
- `jumpIf shape != 3`
- `read for shape 3`
...
Many shapes in practice

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IE</td>
<td>Slow Zone for 2+ shapes</td>
</tr>
<tr>
<td>Opera</td>
<td># of shapes doesn’t matter!</td>
</tr>
<tr>
<td>Chrome</td>
<td>more shapes -&gt; slower</td>
</tr>
<tr>
<td>Firefox</td>
<td>slower with more shapes, but levels off in Slow Zone</td>
</tr>
<tr>
<td>Safari</td>
<td></td>
</tr>
</tbody>
</table>

# of shapes at property read site
Deeply Nested Closures are Slower

- Prototype chains don’t slow us down, but deep closure nesting does. Why?
- Every call to \( f \) generates a **unique** closure object to hold \( x \).
- The engine must **walk** up to \( x \) each time

```
var f = function() {
    var x;
    var g = function() {
        var h = function() {
            var i = function () {
                var j = function() {
                    z = x + y;
                }
            }
        }
    }
}
```
Properties in the Slow Zone

**Undefined Property**
(Fast on Firefox, Chrome)

```javascript
var a = {};
a.x;
```

**DOM Access**
(I only tested .id, so take with a grain of salt--other properties may differ)

```javascript
var a = document.getElementById("foo");
a.id;
```

**Scripted Getter**
(Fast on IE)

```javascript
var a = { x: get() { return 1; } };
a.x;
```

**Scripted Setter**

```javascript
var a = { x: set(y) { this.x_ = y; } };
a.x = 1;
```
The Type-Specializing JIT

Firefox 3.5+
(Tracemonkey)

Chrome 11+
(Crankshaft)
Types FTW!

If only JavaScript had type declarations...

➡ The JIT would know the type of every local variable
➡ Know exactly what action to use (no type checks)
➡ Local variables don’t need to be boxed (or unboxed)

We call this kind of JIT a type-specializing JIT
But JS doesn’t have types

- **Problem**: JS doesn’t have type declarations
  - won’t have them any time soon
  - we don’t want to wait
- **Solution**: run the program for a bit, monitor types
- Then **recompile** optimized for those types
Running with the Type-Specializing JIT

On $x = y + z$:

- read the operation $x = y + z$ from memory
- read the inputs $y$ and $z$ from memory
- check the types of $y$ and $z$ and choose the action
- unbox $y$ and $z$
- execute the action
- box the output $x$
- write the output $x$ to memory
Further Optimization I

Automatic Inlining

original code

```javascript
function getPop(city) {
  return popdata[city.id];
}
for (var i = 0; i < N; ++i) {
total += getPop(city);
}
```

JIT compiles as if you wrote this

```javascript
for (var i = 0; i < N; ++i) {
total += popdata[city.id];
}
```
Further Optimization 2

Loop Invariant Code Motion (LICM, “hoisting”)

**original code**

```javascript
for (var i = 0; i < N; ++i) {
    total += a[i] * (1 + options.tax);
}
```

**JIT compiles as if you wrote this**

```javascript
var f = 1 + options.tax;
for (var i = 0; i < N; ++i) {
    total += a[i] * f;
}
```
Optimize Only Hot Code

- Type-specializing JITs can have a hefty startup cost
  - Need to collect the type information
  - Advanced compiler optimizations take longer to run
- Therefore, type specialization is applied selectively
  - Only on **hot code**
    - Tracemonkey: hot = 70 iterations
    - Crankshaft: hot = according to a profiler
  - Only if judged to be worthwhile (**incomprehensible heuristics**)
Current Limitations

- **What happens if the types change after compiling?**
  - Just a few changes -> recompile, slight slowdown
  - Many changes -> give up and deoptimize to basic JIT

- Array elements, object properties, and closed-over variables
  - Usually still boxed
  - Still need to check type and unbox on get, box on set
  - Typed arrays might help, but support is not always there yet

- JS semantics require overflow checks for integer math
Type Inference for JITs
Type Inference

- Trying to get rid of the last few instances of boxing (from before: array and object properties)
- Idea: use static program analysis to **prove** types
  - of object props, array elements, called functions
  - or, *almost* prove types, and also prove minimal checks needed
Type Inference Example

Type inference gets this...

```
var a = [];
for (var i = 0; i < N; ++i) {
    a[i] = i * i;
}
var sum = 0;
for (var i = 0; i < N; ++i) {
    sum += a[i];
}
```

...but not this.

```
var a = [];
for (var i = 0; i < N; ++i) {
    if (i % 2)
        a[i] = i * i;
    else
        a[i] = "foo";
}
var sum = 0;
for (var i = 0; i < N; ++i) {
    if (i % 2)
        sum += a[i];
}
```

"i is always a number, so i * i is always a number, so a[_] is always a number!"
Type-stable JavaScript

The key to running faster in future JITs is type-stable JavaScript.

This means JavaScript where you could declare a single engine-internal type for each variable.
Type-stable JS: examples

**Type-stable**

```javascript
var g = 34;
var o1 = { a: 56 };
var o2 = { a: 99 };
for (var i = 0; i < 10; ++i) {
    var o = i % 2 ? o1 : o2;
    g += o.a;
}
g = 0;
```

**NOT type-stable**

```javascript
var g = 34;
var o1 = { a: 56 };
var o2 = { z: 22, a: 56 };
for (var i = 0; i < 10; ++i) {
    var o = i % 2 ? o1 : o2;
    g += o.a;  // Different shapes
}
g = "hello";  // Type change
```
Garbage Collection
What Allocates Memory?

**Objects**
- `new Object();`
- `new MyConstructor();`
- `{ a: 4, b: 5 }`
- `Object.create();`

**Arrays**
- `new Array();`
- `[ 1, 2, 3, 4 ];`

**Strings**
- `new String("hello");`
- `"<p>" + e.innerHTML + "</p>"`

**Function Objects**
- `var x = function () { ... }`
- `new Function(code);`

**Closure Environments**
- `function outer(name) {
  var x = name;
  return function inner() {
    return "Hi, " + name;
  }
}

**name is stored in an implicitly created object!**
GC Pauses Your Program!

- Basic GC algorithm (mark and sweep)
  - Traverse all reachable objects (from locals, window, DOM)
  - Recycle objects that are not reachable
- The JS program is **paused** during GC for safe traversal
- **Pauses may be long: 100 ms or more**
  - Serious problem for animation
  - Can also be a drag on general performance
Reducing Pauses with Science 1

**Generational GC**

Idea: Optimize for creating many short-lived objects

Create objects in a frequently collected nursery area
Promote long-lived objects to a rarely collected tenured area

Chrome

Mozilla
Generational GC by Example

**scavenging** young generation (aka nursery)

- Point
- Point
- Line
- Point
- Message
- Point

**mark-and-sweep** tenured generation

- Message
- Array
- Message

mozilla
Reducing Pauses with Science II

**Incremental GC**

Idea: Do a little bit of GC traversal at a time

- **Simple GC**
  - JavaScript Running
  - GC Running
  - JS Paused

- **Incremental GC**
  - shorter pauses!

Current Research @Mozilla
Reducing Pauses in Practice

- For all GCs
  - Fewer live objects -> shorter pauses (if not incremental), less time spent in GC
- For simple GCs
  - Lower allocation rate (objects/second) -> less frequent pauses
- For generational GCs
  - Short-lived objects don’t affect pause frequency
  - Long-lived objects cost extra (promotion = copying)
JavaScript Engines in Practice
Performance Faults

- **Performance fault**: when a tiny change hurts performance
  - Sometimes, just makes one statement slower
  - Other times, deoptimizes the entire function!

- **Reasons** we have performance faults
  - **bug**, tends to get quickly
  - "rare" case, will get fixed if not rare
  - **hard to optimize**, RSN...
Strings

• In the Slow Zone, but some things are faster than you might think
  • `.substring()` is fast, \(O(1)\)
  • Don’t need to copy characters, just point within original
  • Concatenation is also optimized
    • Batch up inputs in a **rope** or **concat tree**, concat all at once
  • Performance fault: prepending (Chrome, Opera)

```javascript
// Prepending example
var s = "";
for (var i = 0; i < 100; ++i) {
  s = i + s;
}
```
Arrays

**fast: dense array**

```javascript
var a = [];  
for (var i = 0; i < 100; ++i) {
  a[i] = 0;
}
```

**3-15x slower: sparse array**

```javascript
var a = [];  
a[10000] = 0;  
for (var i = 0; i < 100; ++i) {
  a[i] = 0;
}  
a.x = 7; // Fx, IE only
```

**Want a fast array?**

- Make sure it’s **dense**
- 0..N fill or push fill is always dense
- Huge gaps are always sparse
- N..0 fill is sparse on Firefox
- adding a named property is sparse on Firefox, IE
Iteration over Arrays

**fastest: index iteration**

```javascript
// This runs in all in JIT code,
// so it's really fast.
for (var i = 0; i < a.length; ++i) {
  sum += a[i];
}
```

**3-15x slower: functional style**

```javascript
// This makes N function calls,
// and most JITs don’t optimize
// through C++ reduce().
sum = a.reduce(function(a, b) {
  return a + b;
});
```

**20-80x slower: for-in**

```javascript
// This calls a C++ function to
// navigate the property list.
for (var i in a) {
  sum += a[i];
}
```
Functions

• Function calls use ICS, so they are fast
  • Manual inlining can still help sometimes

• Key performance faults:  
  • `f.call()` - 1.3-35x slower than `f()`
  • `f.apply()` - 5-50x slower than `f()`
  • arguments - often very slow, but varies
Creating Objects

Creating objects is slow
Doesn’t matter too much how you create or populate

Exception: Constructors on Chrome are fast

```javascript
function Cons(x, y, z) {
    this.x = x;
    this.y = y;
    this.z = z;
}

for (var i = 0; i < N; ++i)
    new Cons(i, i + 1, i * 2);
```
OOP Styling

Prototype

```javascript
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype = {
    distance: function(pt2) ...
}
```

Prototype style is much faster to create (each closure creates a function object)

Information-Hiding

```javascript
function Point(x, y) {
    return {
        distance: function(pt2) ...
    }
}
```

Instance Methods

```javascript
function Point(x, y) {
    this.x = x;
    this.y = y;
    this.distance = function(pt2) ...
}
```

Using the objects is about the same
Exceptions

- Exceptions assumed to be rare in perf-sensitive code
  - running a try statement is free on most browsers
  - throw/catch is really slow
- There are many performance faults around exceptions
  - just having a try statement deoptimizes on some browsers
  - try-finally is perf fault on some
eval and with

Short version:
Do not use anywhere near performance sensitive code!

**Mind-Bogglingly Awful**
5-100x slower than using a function call

```javascript
var sum = 0;
for (var i = 0; i < N; ++i) {
    sum = eval("sum + i");
}
```

**Still Terrible**
2-10x slower than without eval

```javascript
var sum = 0;
eval("\n");
for (var i = 0; i < N; ++i) {
    sum = eval("sum + i");
}
```
Top 5 Things to Know

5. Avoid eval, with, exceptions near perf-sensitive code
4. Avoid creating objects in hot loops
3. Use dense arrays (know what causes sparseness)
2. Write type-stable code
1. ....
Talk To Us

**JS engine developers want to help you. Tell us about:**

- Performance faults you run into
- Exciting apps that require fast JS
- Anything interesting you discover about JS performance