Improving JavaScript Performance by Deconstructing the Type System

Wonsun Ahn, Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas
University of Illinois at Urbana-Champaign
PLDI 2014
Motivation:
Poor JavaScript Performance on Websites

• JSBench: Package of existing websites

Industry Benchmarks

Normalized Instructions

- Chrome V8
- Chrome V8 (Optimizations Disabled)
Contributions

• Detailed characterization of V8 behavior while running:
  – JSBench (popular websites)
  – Industry benchmarks

• Identified reasons behind poor performance on websites
  – Frequent type specialization failure due to rampant type creation
  – Due to encoding inheritance and method bindings in the type system

• Modified V8:
  – 36% in performance improvement
  – 49% in dynamic instruction count reduction
  – 20% in heap memory usage reduction
Why Traditional Languages are Fast: Type Declarations

- Types are crucial to generating efficient code

[C Code]
```
struct A {
    int x;  // offset 0
    int y;  // offset 1
};
A o;
o.x = 11;
o.y = 22;
```

[Assembly Code]
```
// o.x = 11;
store &o[0], 11

// o.y = 22;
store &o[1], 22
```

Types tell compiler the shape of o (fields and their offsets)
Scripting Languages Have No Types

- Objects are simply dictionaries from properties to values
- Properties can be added and removed at any time

```
[JavaScript Code]
var o;
if (...) {
    o.x = 11;
}
o.y = 22;
```

Compile
Shape of \( o \)?

[Object \( o \)]
\( y : 22 \)

(If branch not taken)

OR

[Object \( o \)]
\( x : 11 \)
\( y : 22 \)

(If branch taken)

How to generate code when shape of \( o \) is unknown?
Scripting Languages Have No Types

- Objects are simply dictionaries from properties to values
- Properties can be added and removed at any time

[JavaScript Code]
```javascript
var o;
if (...) {
    o.x = 11;
}
o.y = 22;
```

Compile

Hash tables

Shape of `o`?

[Object `o`]
- Property `y`: Offset 0
- Property `type_ptr`: Offset 0

[Type A]
- Property `y`: Offset 0

[Object `o`]
- Property `x`: Offset 0
- Property `y`: Offset 1
- Property `type_ptr`: Offset 0

[Type B]
- Property `x`: Offset 0
- Property `y`: Offset 1

Compilers introduce a type system behind the scenes.
Scripting Languages Have No Types → Slow

[Object o]
y : 22
   type_ptr

[Type A]
<table>
<thead>
<tr>
<th>Property</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>

OR

[Object o]
x : 11
  y : 22
     type_ptr

[Type B]
<table>
<thead>
<tr>
<th>Property</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>

[JavaScript Code]
var o;
if (...) {
  o.x = 11;
}
o.y = 22;

[Compile]
offset = lookup (o.type_ptr, "y");
store &o[offset], 22;

A field access always entails a hash table lookup to get the offset
State-of-the-art Compilers do Better:
Type Specialization Optimization

- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code

[JavaScript Code]
```javascript
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

[Type A]
```
<table>
<thead>
<tr>
<th>Property</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>
```

[Type B]
```
<table>
<thead>
<tr>
<th>Property</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>
```

[Object o]
y : 22  
type_ptr

[Object o]
x : 11  

[Object o]
y : 22  
type_ptr

[Type A] OR

[Type B]

[Pseudo Code for o.y = 22]
```javascript
if (o.type_ptr == A)
  offset = 0;
else
  offset = lookup (o.type_ptr, “y”);
store &o[offset], 22
```
State-of-the-art Compilers do Better: 
**Type Specialization Optimization**

- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types \(\Rightarrow\) Fast

---

**JavaScript Code**
```javascript
var o;
if (...) {
    o.x = 11;
}
o.y = 22;
```

**Pseudo Code for \(o.y = 22\)**
```plaintext
If (\(o\_type\_ptr == A\))
    offset = 0;
else
    offset = lookup (\(o\_type\_ptr, \"y\")
store &o[offset], 22
```

---

**Object**
- **Type A**
  - **Property**: \(x\)
  - **Offset**: 0
  - **Property**: \(y\)
    - **Offset**: 0
  - **Property**: \(type\_ptr\)

- **Type B**
  - **Property**: \(x\)
    - **Offset**: 0
  - **Property**: \(y\)
    - **Offset**: 1
State-of-the-art Compilers do Better: Type Specialization Optimization

- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types ➔ Fast
- **IC Miss**: Execution results in a hash table lookup ➔ Very Slow

---

**[JavaScript Code]**
```javascript
var o;
if (...) {
  o.x = 11;
} 
o.y = 22;
```

**[Object o]**
- **y**: 22
  - `type_ptr`

**[Type A]**
- **y**: Offset 0

**[Object o]**
- **x**: 11
  - `type_ptr`

**[Type B]**
- **x**: Offset 0
- **y**: Offset 1

---

**[Pseudo Code for o.y = 22]**
```pseudo
If (o.type_ptr == A)
  Offset = 0;
else
  Offset = lookup (o.type_ptr, “y”);
store &o[Offset], 22
```

---

---

Fast and Secure Mobile Computing
State-of-the-art Compilers do Better: Type Specialization Optimization

- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types → Fast
- **IC Miss**: Execution results in a hash table lookup → Very Slow

[JavaScript Code]
```javascript
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

[Compile]
```c
[Object o]
  y : 22
  type_ptr

[Type A]
  Property   Offset
  y           0

[Object o]
  x : 11
  type_ptr

[Type B]
  Property   Offset
  x           0
  y           1
```

[IC Miss]
```
[Pseudo Code for o.y = 22]
If (o.type_ptr == A)
  offset = 0;
else if (o.type_ptr == B)
  offset = 1;
else
  offset = lookup (o.type_ptr, “y”);
store &o[offset], 22
```

Fast and Secure Mobile Computing
State-of-the-art Compilers do Better:
Type Specialization Optimization

- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types \(\Rightarrow\) Fast
- **IC Miss**: Execution results in a hash table lookup \(\Rightarrow\) Very Slow

---

**Compile**

**[JavaScript Code]**
```javascript
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

**[Object o]**
- `y : 22`
- `type_ptr`

**[Type A]**
- `y : 0`

**[Object o]**
- `x : 11`
- `type_ptr`

**[Type B]**
- `x : 0`
- `y : 1`

**[Pseudo Code for o.y = 22]**
```pseudo
If (o.type_ptr == A)
  offset = 0;
else if (o.type_ptr == B)
  offset = 1;
else
  offset = lookup (o.type_ptr, “y”);
store &o[offset], 22
```

---

**IC Hit**

- **Property Offset**
  - `y : 0`
  - `x : 1`

---

**Fast and Secure Mobile Computing**
Google Chrome V8 JavaScript Compiler: Ineffective for Many Real Websites

Normalized Instructions

Websites

Industry Benchmarks

Chrome V8 compiler not optimized for dynamism in real websites
Problem: Chrome V8 Type System is Too Brittle

- Chrome V8 type system encodes (other than properties):
  - **Inheritance** (i.e. address of parent object)
  - **Method bindings** (i.e. addresses of functions called)
- Helps in generating efficient code during type specialization
  - Inheritance: helps when accessing parent properties
  - Method bindings: helps resolve targets for method calls
- V8 Assumption: inheritance and method bindings rarely change
  - Reasonable since always true for statically typed languages

---

Reality: Assumption **NOT TRUE** for dynamic website code

- Leads to type dynamism and frequent inline cache misses (and terrible performance)
- This is the basis of our optimization
Inheritance in JavaScript: Prototype Objects

• C++ or Java uses types to do inheritance
  – Using parent classes and child classes
• JavaScript has no types ➔ uses prototype objects
• Prototype objects
  – Can be thought of “parent objects”
  – Serve similar purposes as parent classes
  – Regular objects just like all other objects
Inheritance in JavaScript: Prototype Objects

1. Create function `foo` (implicitly creates “parent” `foo.prototype`)
2. Create “child” `o` by calling `foo` (o inherits from `foo.prototype`)
3. Access “parent” property `x` in `foo.prototype` through inheritance
How V8 Encodes Inheritance into Types

1. Create type that inherits `foo.prototype` (by linking through `parent_ptr`)
2. Have all objects created by `foo()` have that type
3. Access parent properties through `type_ptr` and `parent_ptr` links

// Create “parent” object
var foo = function () { ... };
foo.prototype.x = 11;
// Create “children” objects
var o = new foo();
var o2 = new foo();
// Access “parent” property
print(o.x); // 11
Problem: Rampant Type Creation
Due to Encoding Inheritance

```javascript
while (...) {
    var foo = function () { ... };
    foo.prototype.x = 11;

    var o = new foo();
    print(o.x); // 11
}
```

[Inline Cache for load o.x]
/* hash table access */
Problem: Rampant Type Creation Due to Encoding Inheritance

while (...) {
    var foo = function () { ... };  
    foo.prototype.x = 11;
    var o = new foo();
    print(o.x); // 11
}

[Inline Cache for load o.x]
/* hash table access */

[Object foo.prototype]
x : 11

[Type foo]
parent_ptr

[Object o]
type_ptr

IC Miss on foo
Problem: Rampant Type Creation Due to Encoding Inheritance

```javascript
while (...)
  var foo = function () {
    foo.prototype.x = 11;
    var o = new foo();
    print(o.x); // 11
  }
```

**[Inline Cache for load o.x]**
If (o.type_ptr == foo₁) {
  prototype = foo₁.proto_ptr;
  // access “x” in prototype
} else {
  /* miss handling */
}

**Iteration 1**

**[Object foo.prototype]**

```
x : 11
```

**[Type foo₁]**

```
parent_ptr
```

**[Object o]**

```
type_ptr
```

IC Miss on foo₁
Problem: Rampant Type Creation Due to Encoding Inheritance

while (...) {
  var foo = function () { ... };  
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}

[Inline Cache for load o.x]
If (o.type_ptr == foo₁) {
  prototype = foo₁.proto_ptr;
  // access “x” in prototype
} else {
  /* miss handling */
}

IC Miss on foo₂; Add foo₂ to IC
Repeated on all future iterations
Problem: Rampant Type Creation
Due to Encoding Inheritance

```
while (...) {
    var foo = function () { ... };  
    foo.prototype.x = 11;
    var o = new foo();
    print(o.x); // 11
}
```

```
var foo = function () {
    ...
};
foo.prototype.x = 11;
var o = new foo();
print(o.x); // 11
```

```
while (...)
```

```
while (...) {
    var foo = function () { ...};
    foo.prototype.x = 11;
    var o = new foo();
    print(o.x); // 11
}
```

```
while (...) {
    var foo = function () { ...};
    foo.prototype.x = 11;
    var o = new foo();
    print(o.x); // 11
}
```

- Very counterintuitive: same code creates different types
- Dynamic function creation: common pattern in website code
  - For encapsulation and ease of programming
Solution: Decouple Inheritance from Types

```javascript
while (...) {
    var foo = function () { ... };
    foo.prototype.x = 11;

    var o = new foo();
    print(o.x); // 11
}

[Inline Cache for load o.x]
/* hash table access */
```
Solution: Decouple Inheritance from Types

while (...) {
    var foo = function () { ... };
    foo.prototype.x = 11;

    var o = new foo();
    print(o.x); // 11
}

/* hash table access */
Solution: Decouple Inheritance from Types

while (...) {
  var foo = function () { ... };  
  foo.prototype.x = 11;
  var o = new foo();
  print(o.x);  // 11
}

/* hash table access */

Iteration 1

[Object foo.prototype]
  x: 11

[Type foo]

[Object o]
  type_ptr
  parent_ptr

[Inline Cache for load o.x]
Solution: Decouple Inheritance from Types

while (...) {
    var foo = function () { ... };  
    foo.prototype.x = 11;
    var o = new foo();
    print(o.x); // 11
}

/* hash table access */

Iteration 1

[Object foo.prototype]
  x : 11

[Type foo₁]

[Object o]
  type_ptr
  parent_ptr

IC Miss on foo₁

Fast and Secure Mobile Computing
Solution: Decouple Inheritance from Types

while (...) {
  var foo = function () { ... };  
  foo.prototype.x = 11;

  var o = new foo();
  print(o.x); // 11
}

[Inline Cache for load o.x]
If (o.type_ptr == foo₁) {
  prototype = o.proto_ptr;
  // access “x” in parent
} else { /* miss handling */ }

Fast and Secure Mobile Computing
Solution: Decouple Inheritance from Types

```javascript
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;
  var o = new foo();
  print(o.x); // 11
}
```

[Inline Cache for load o.x]
If (o.type_ptr == foo₁) {
  prototype = o.proto_ptr;
  // access “x” in parent
} else { /* miss handling */ }

**Iteration 1**
- `[Object foo.prototype]` with `x : 11`
- `[Type foo₁]`
- `[Object o]` with `type_ptr parent_ptr`

**Iteration 2**
- `[Object foo.prototype]` with `x : 11`
- `[Object o]` with `type_ptr parent_ptr`

IC Hit on `foo₁`
Repeated on all future iterations
Solution: Decouple Inheritance from Types

while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;
  var o = new foo();
  print(o.x); // 11
}

[Inline Cache for load o.x]
If (o.type_ptr == foo₁) {
  prototype = o.proto_ptr;
}

Iteration 1

- [Object foo.prototype]
  x : 11

Iteration 2

- [Object foo.prototype]
  x : 11

- [Object o]
  type_ptr
  parent_ptr

- [Type foo₁]

• Together with method binding decoupling, removes almost all inline cache misses and most type polymorphism
Problem: Rampant Type Creation Due to Encoding Method Bindings

```javascript
var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
  while (...) {
    var o = new foo();
  }
```

### Iteration 1

- **Type foo₀**:
  - Prop: bar, Offset: 
  - Prop: baz, Offset: 

### Iteration 2

- **Type foo₁**:
  - Prop: bar, Offset: 0
- **Type foo₂**:
  - Prop: bar, Offset: 0
  - Prop: baz, Offset: 

### Iteration 3

- **Type foo₃**:
  - Prop: bar, Offset: 0
- **Type foo₄**:
  - Prop: bar, Offset: 0
  - Prop: baz, Offset: 1

---

Improving JavaScript Performance
Problem: Rampant Type Creation
Due to Encoding Method Bindings

var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
} while (...) {
  var o = new foo();
}

- New type at end of each iteration, during the first $N + 1$ iterations
  - Where $N$ is the number of function properties updated
- $O(N^2)$ type objects created in total

Improving JavaScript Performance
Solution: Decouple Method Bindings from Types

var foo = function() {
  this.bar = function() {
    ...
  }
  this.baz = function() {
    ...
  }
} while (...) {
  var o = new foo();
}
var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
} while (...) {
  var o = new foo();
}
Experimental Setup

- Modified V8 Full Compiler with type decoupling enhancement

- Benchmark Suites: JSBench (websites), Kraken, Octane, Sunspider
  - 3 warm-ups before taking measurements to get steady state

- Measurements
  - Xeon machine (performance), Pin tool (characterization)

- Tested three configurations:
  - **B**: Baseline Chrome V8
  - **P**: Prototype inheritance decoupling
  - **C**: Combined inheritance & method binding decoupling
Instruction Breakdown for Chrome V8

- JSBench: 50% of instructions spent on inline cache miss handling
- Kraken, Octane, Sunspider: Most instructions spent on generated code
Performance Improvement over Chrome V8

- Final execution time decreased by 36% compared to Chrome V8
Instruction Count Reduction over Chrome V8

- Almost all instructions due to inline cache misses removed
- Total instruction count decreased by 49% compared to Chrome V8
- Negligible change for Kraken, Octane, Sunspider (not shown)
Summary

• In website JavaScript code, rampant creation of types causes
  – Performance overhead due to frequent inline cache misses
  – Performance overhead due to inline cache bloat
    (i.e. multiple if statements for multiple types)
  – Code memory bloat due to inline cache bloat
  – Heap memory bloat due to excessive type object creation
    ➔ Reason: Google Chrome V8 type system is too brittle

• Made the type system of Google Chrome V8 more flexible:
  – By decoupling inheritance and method binding from types

Results in:
  36% in performance improvement
  49% in dynamic instruction count reduction
  20% in heap memory usage reduction
Improving JavaScript Performance by Deconstructing the Type System

Wonsun Ahn, Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas
University of Illinois at Urbana-Champaign
PLDI 2014

Please direct questions at: dahn2@illinois.edu