Improving JavaScript Performance by Deconstructing the Type System

Wonsun Ahn, Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas

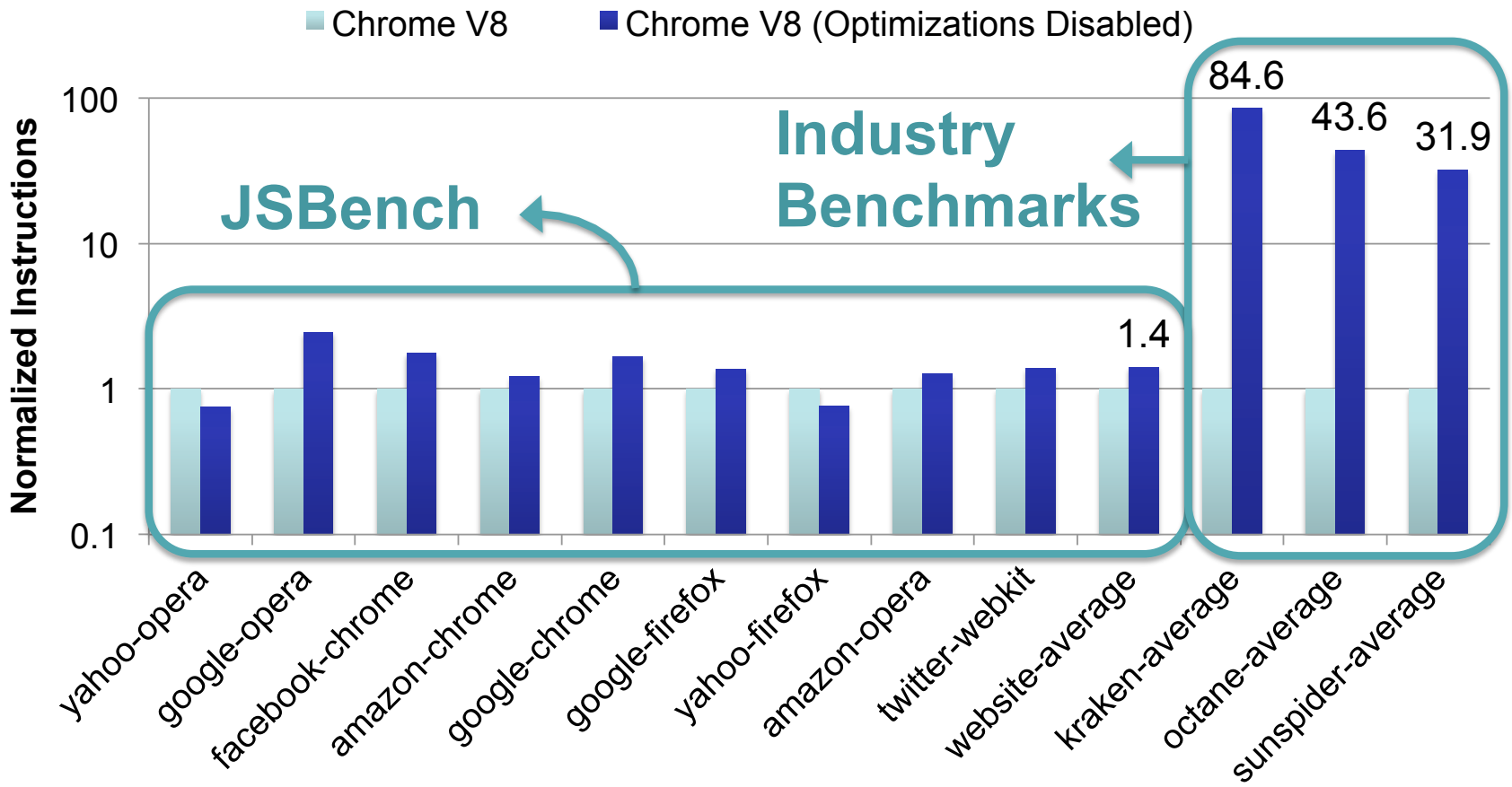
University of Illinois at Urbana-Champaign

PLDI 2014

i-acoma group

ILLINOIS

Motivation:
Poor JavaScript Performance on Websites



- JSBench: Package of existing websites

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;
```

Compile →

[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

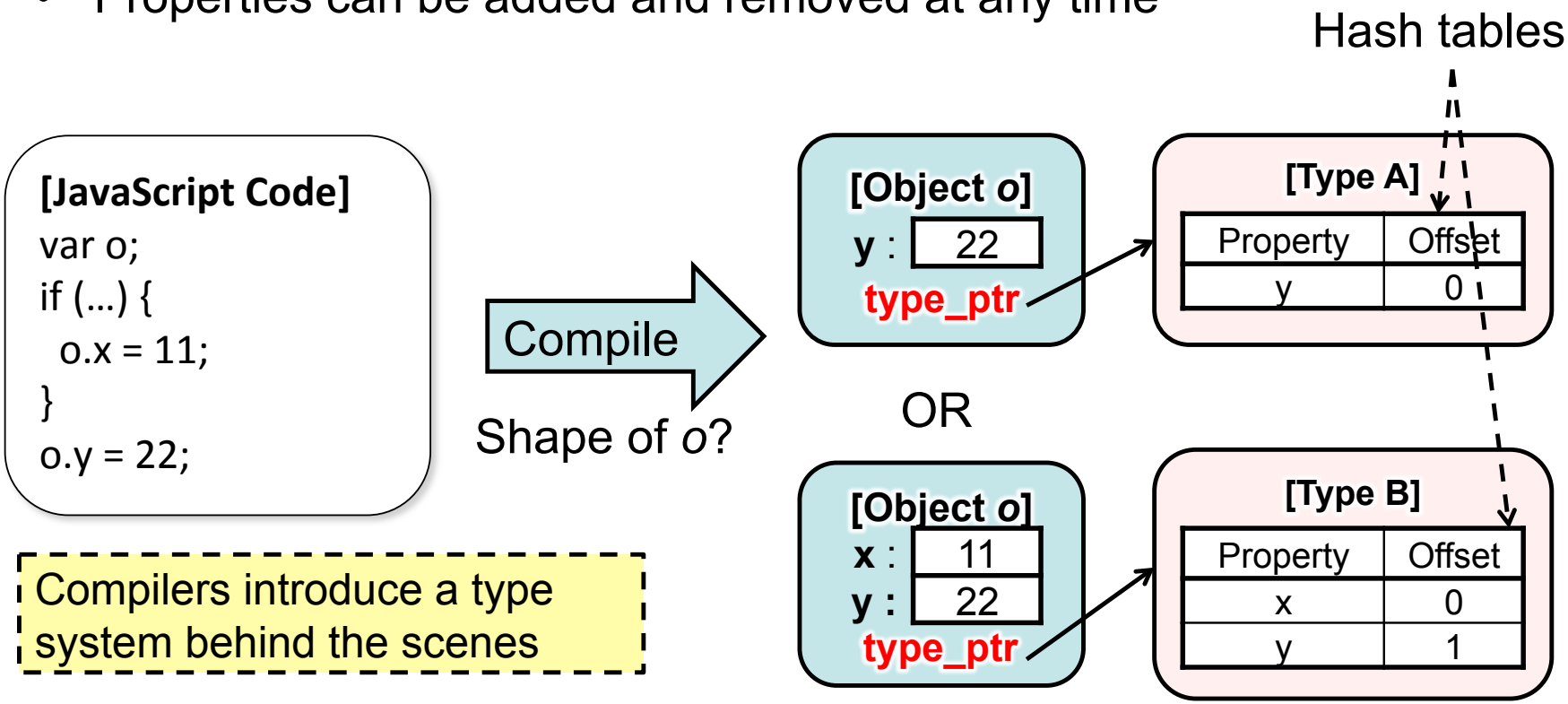
[Object *o*]
x : 11
y : 22

(If branch taken)

How to generate code when shape of *o* is unknown?

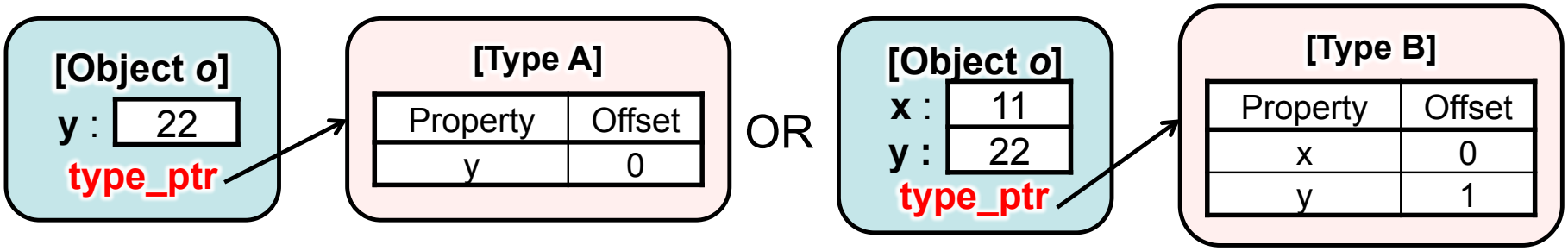
Scripting Languages Have No Types

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



Compilers introduce a type system behind the scenes

Scripting Languages Have No Types ➔ Slow



[JavaScript Code]

```
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

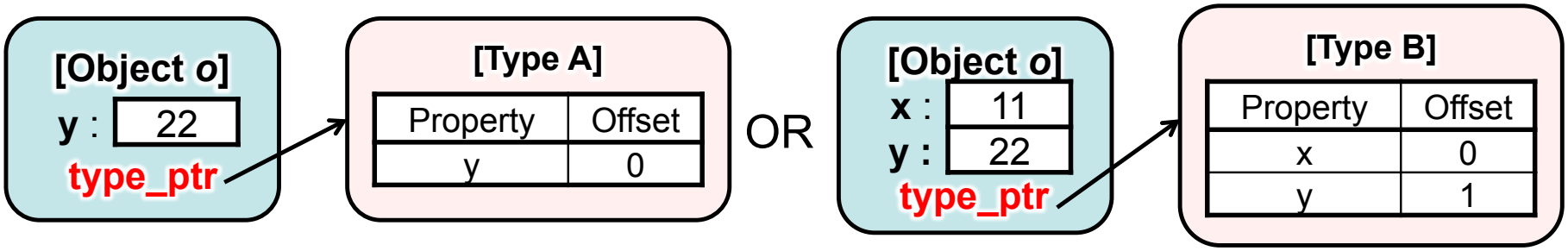
Compile ➔

[Pseudo Code for *o.y = 22*]

```
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



[JavaScript Code]

```
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

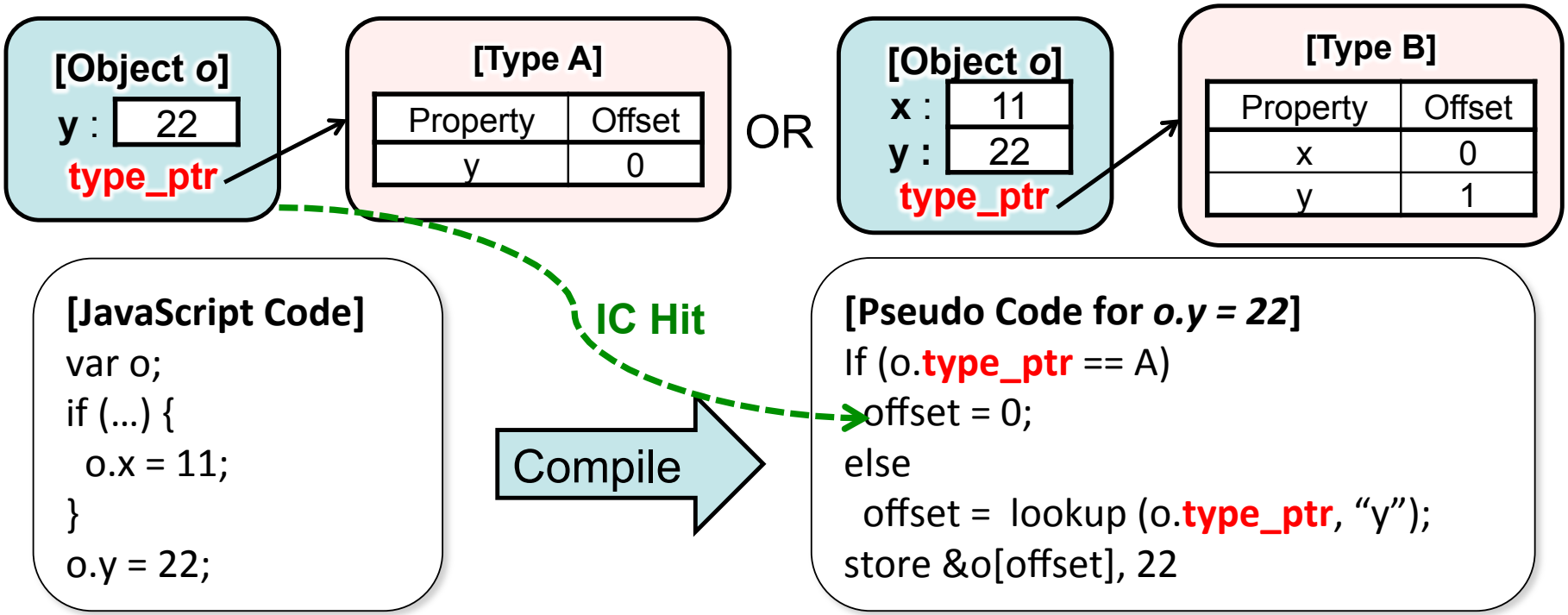
Compile

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else
  offset =  lookup (o.type_ptr, "y");
store &o[offset], 22
```

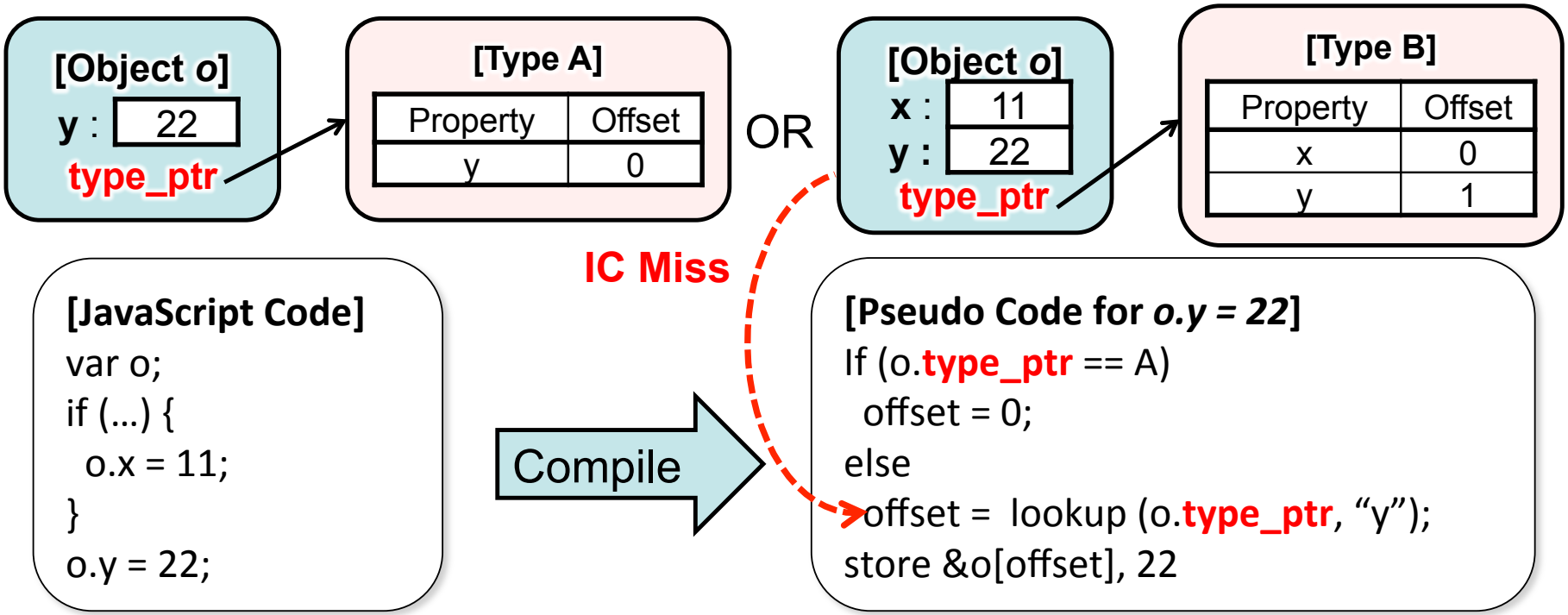
- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code

State-of-the-art Compilers do Better: Type Specialization Optimization



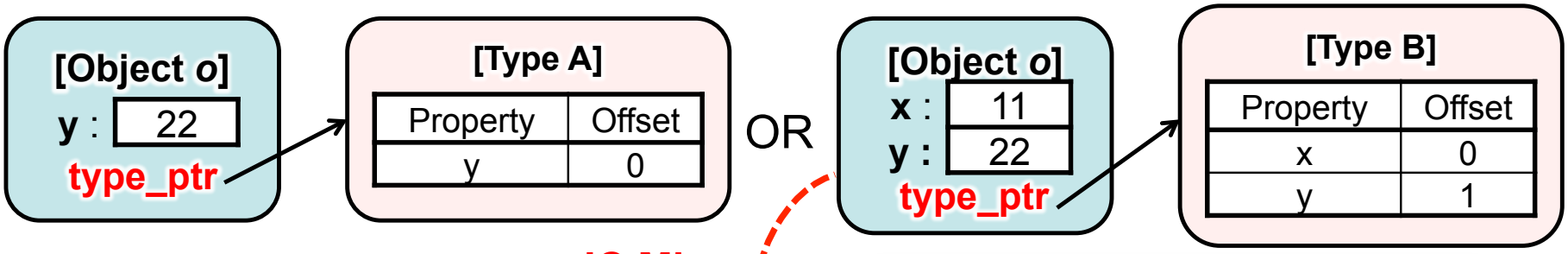
- **Type specialization**: Optimizing code to be fast for the recorded types
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types ➔ Fast

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

State-of-the-art Compilers do Better: Type Specialization Optimization



IC Miss

[JavaScript Code]

```
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

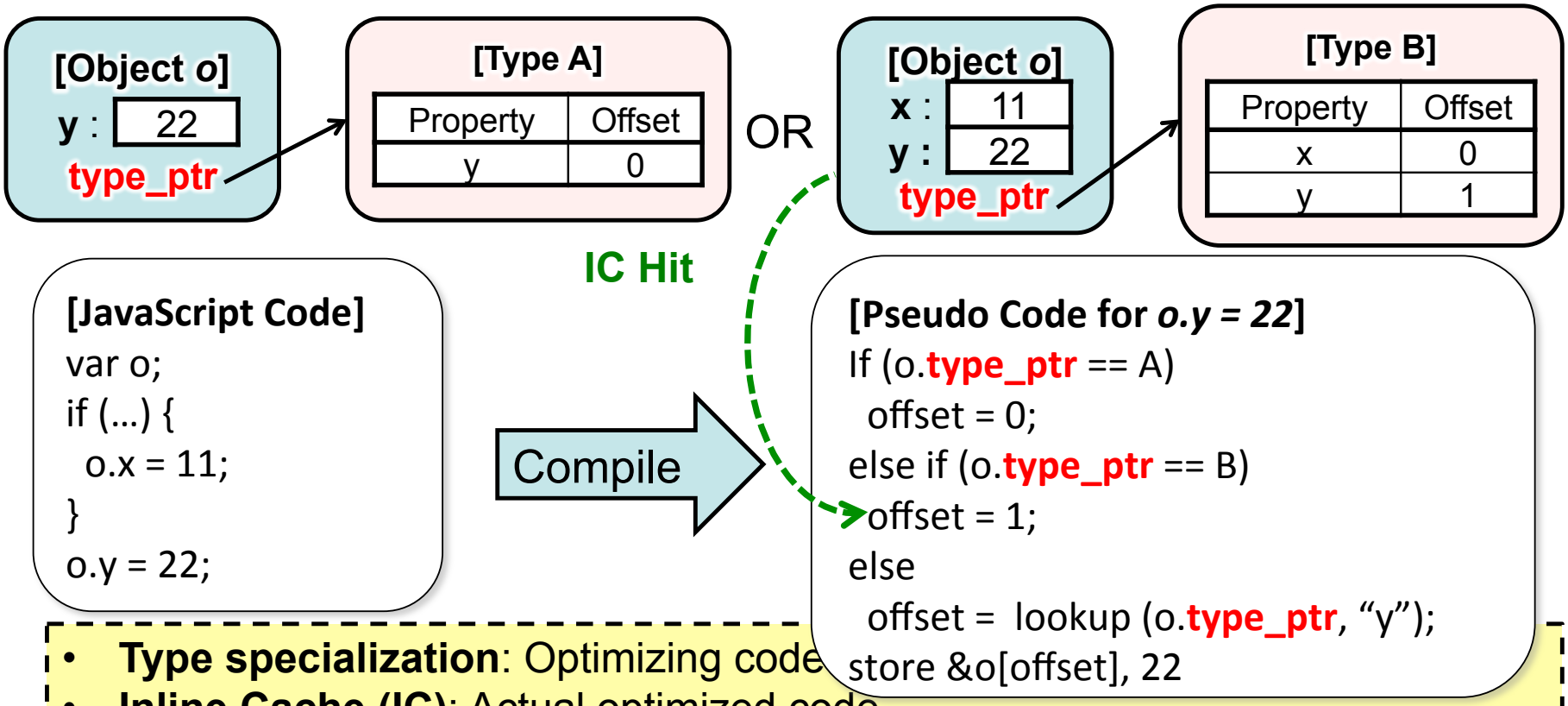
Compile

[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
```

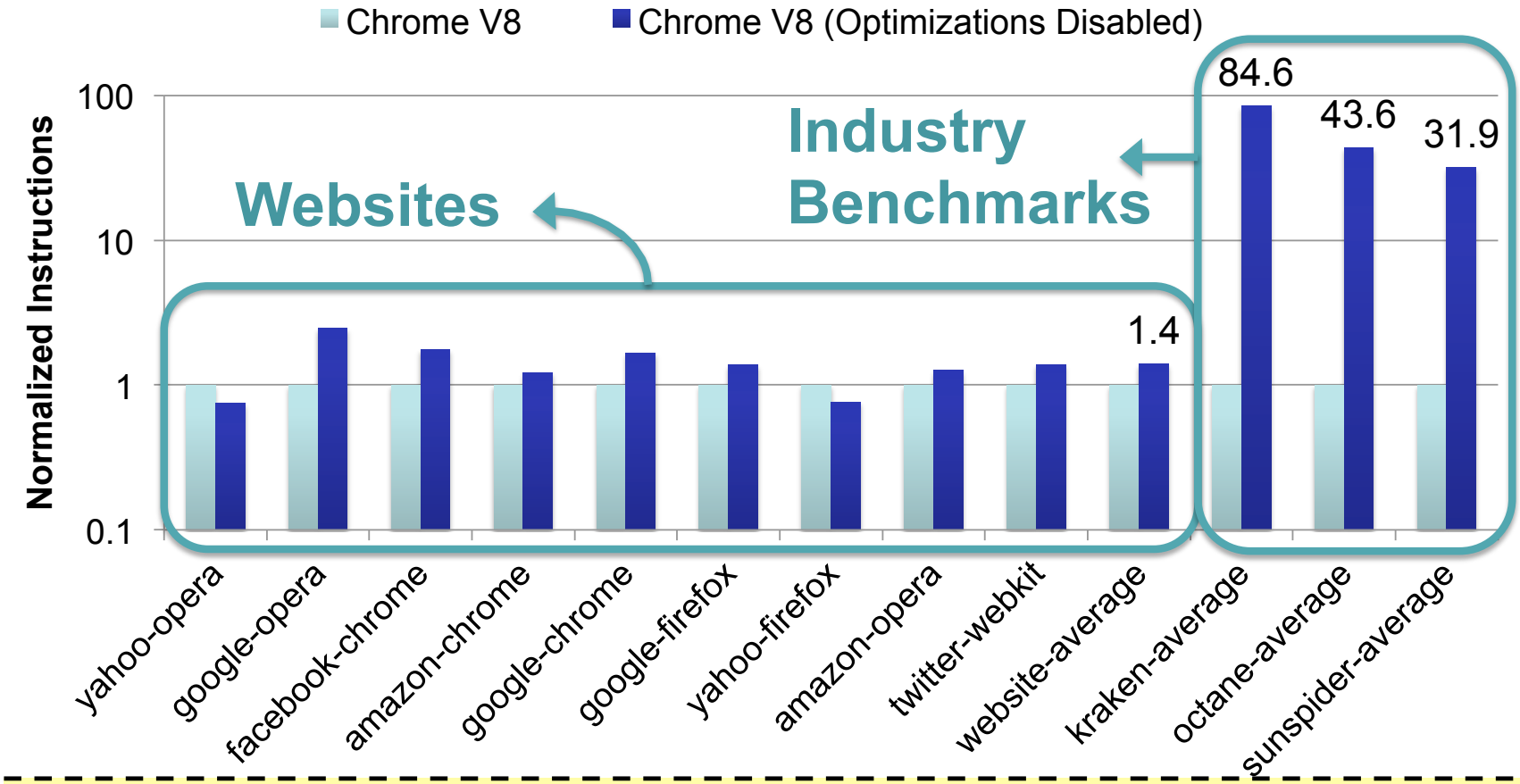
- **Type specialization**: Optimizing code
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types ➔ Fast
- **IC Miss**: Execution results in a hash table lookup ➔ Very Slow

State-of-the-art Compilers do Better: Type Specialization Optimization



- **Type specialization**: Optimizing code
- **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

Google Chrome V8 JavaScript Compiler: Ineffective for Many Real Websites



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

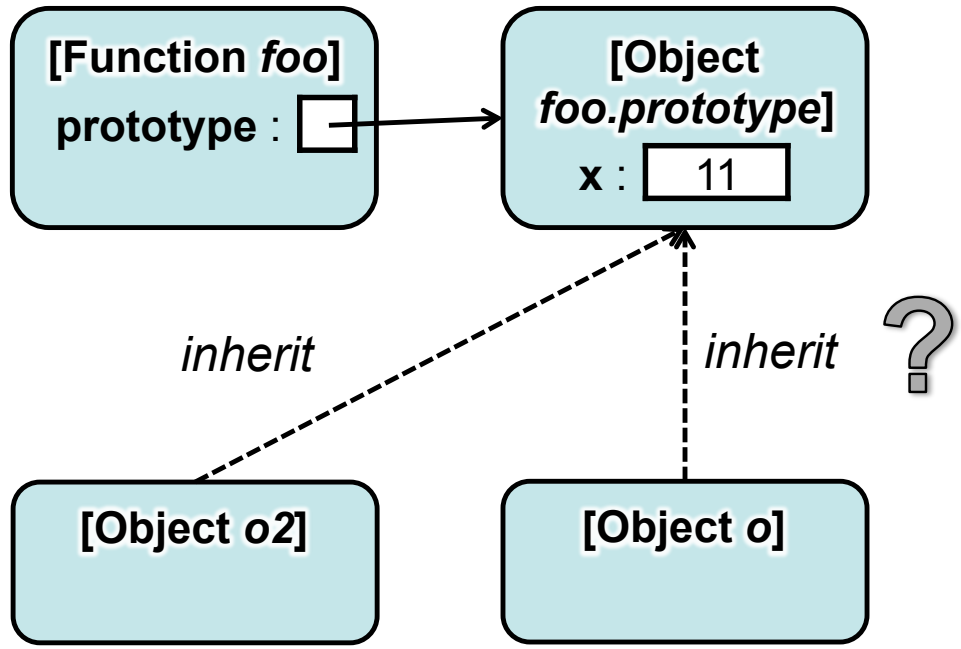
Reality: Assumption **NOT TRUE** for dynamic website code
➜ Leads to type dynamism and frequent inline cache misses (and terrible performance)
➜ This is the basis of our optimization

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

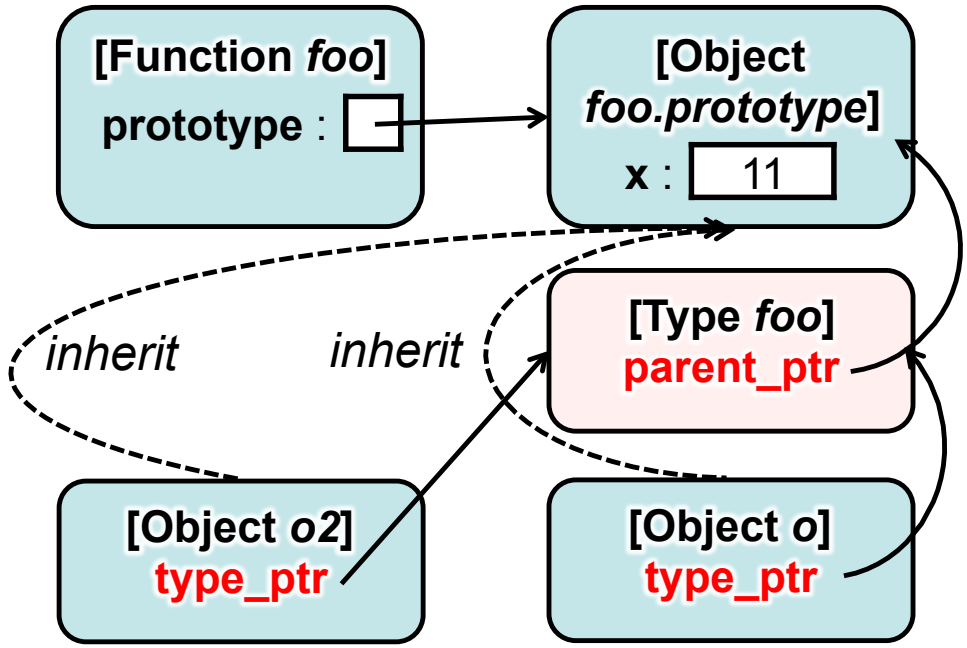
```
// Create "parent" object
var foo = function () { … };
foo.prototype.x = 11;
// Create "children" objects
var o = new foo();
var o2 = new foo();
// Access "parent" property
print(o.x); // 11
```



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

```
// 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
```



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

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 */

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 */

Iteration 1

[Object *foo.prototype*]
x : 11

[Type foo_1]
parent_ptr

[Object *o*]
type_ptr

IC Miss on foo_1

Problem: Rampant Type Creation Due to Encoding Inheritance

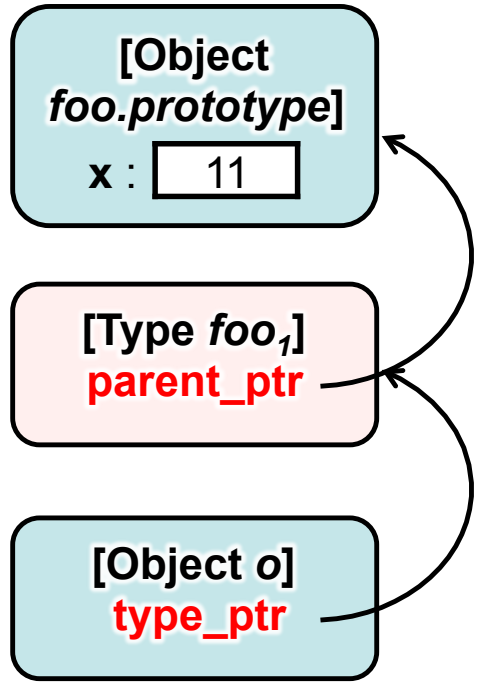
```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  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 */ }
```

Iteration 1



IC Miss on foo_1

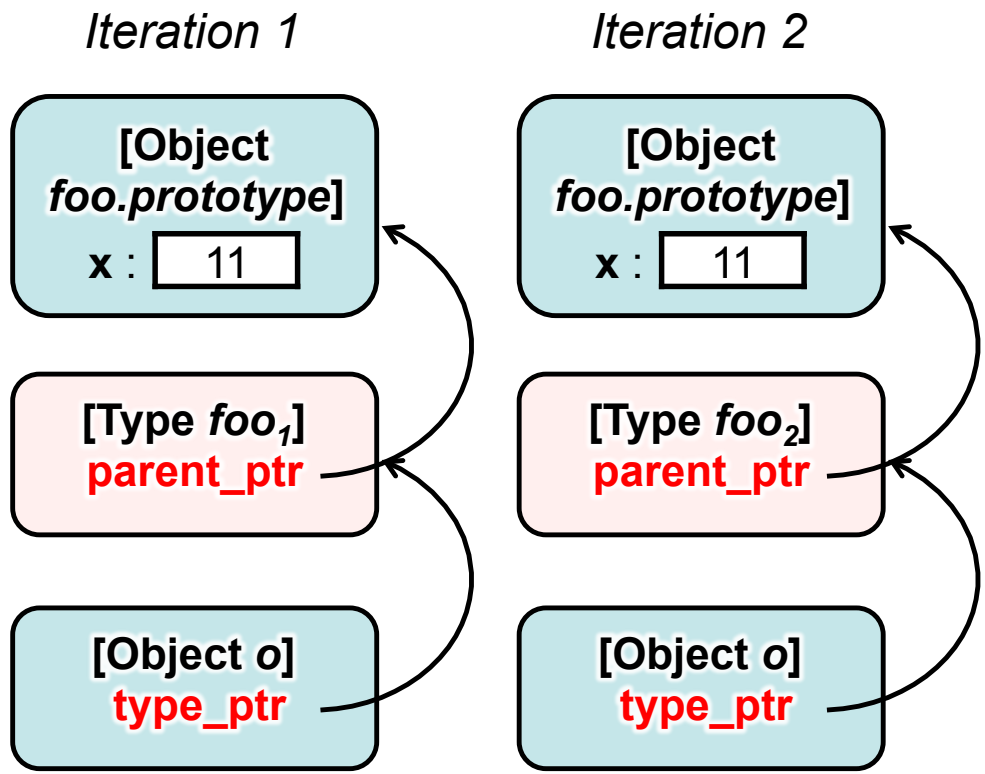
Problem: Rampant Type Creation Due to Encoding Inheritance

```
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 foo_2; Add foo_2 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
}
```

[Inline Cache for *load o.x*]

If (o.type_ptr == foo_1) {

Iteration 1

[Object *foo.prototype*]
x : 11

[Type foo_1]
parent_ptr

[Object *o*]
type_ptr

Iteration 2

[Object *foo.prototype*]
x : 11

[Type foo_2]
parent_ptr

[Object *o*]
type_ptr

- 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

```
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 */

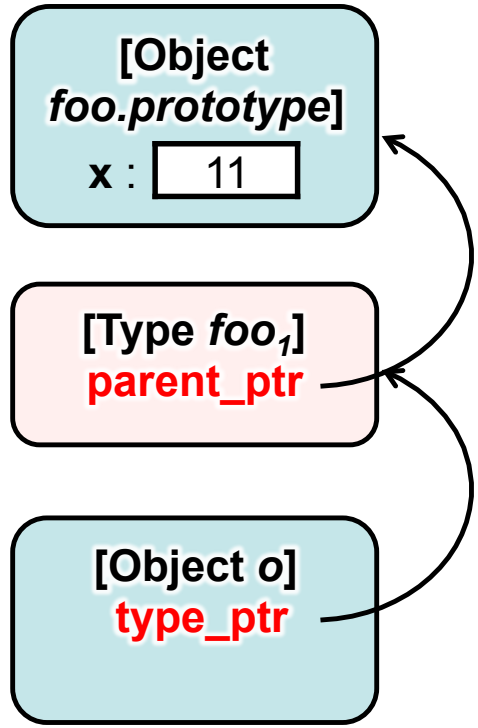
Solution: Decouple Inheritance from Types

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

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

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

Iteration 1



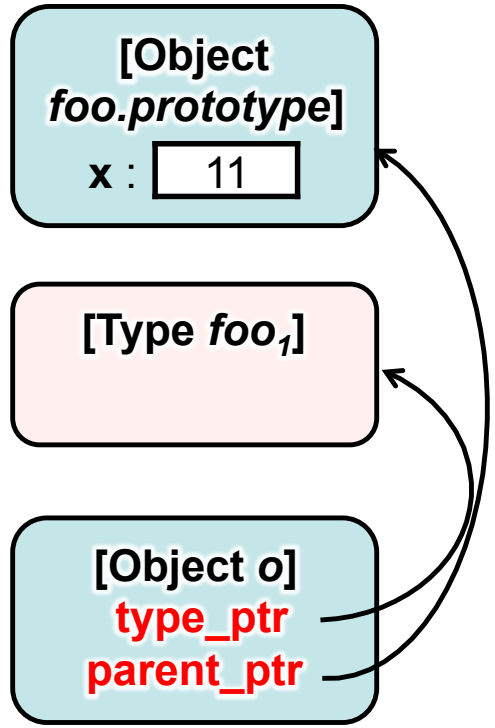
Solution: Decouple Inheritance from Types

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

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

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

Iteration 1



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*]
/* hash table access */

Iteration 1

[Object *foo.prototype*]
x : 11

[Type foo_1]

[Object *o*]
type_ptr
parent_ptr

IC Miss on foo_1

Solution: Decouple Inheritance from Types

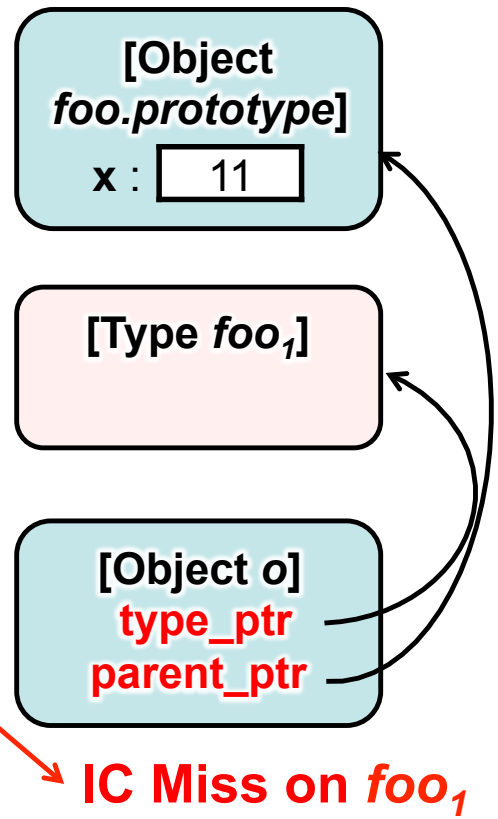
```
while (...) {
  var foo = function () { ... };
  foo.prototype.x = 11;

  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 */ }
```

Iteration 1



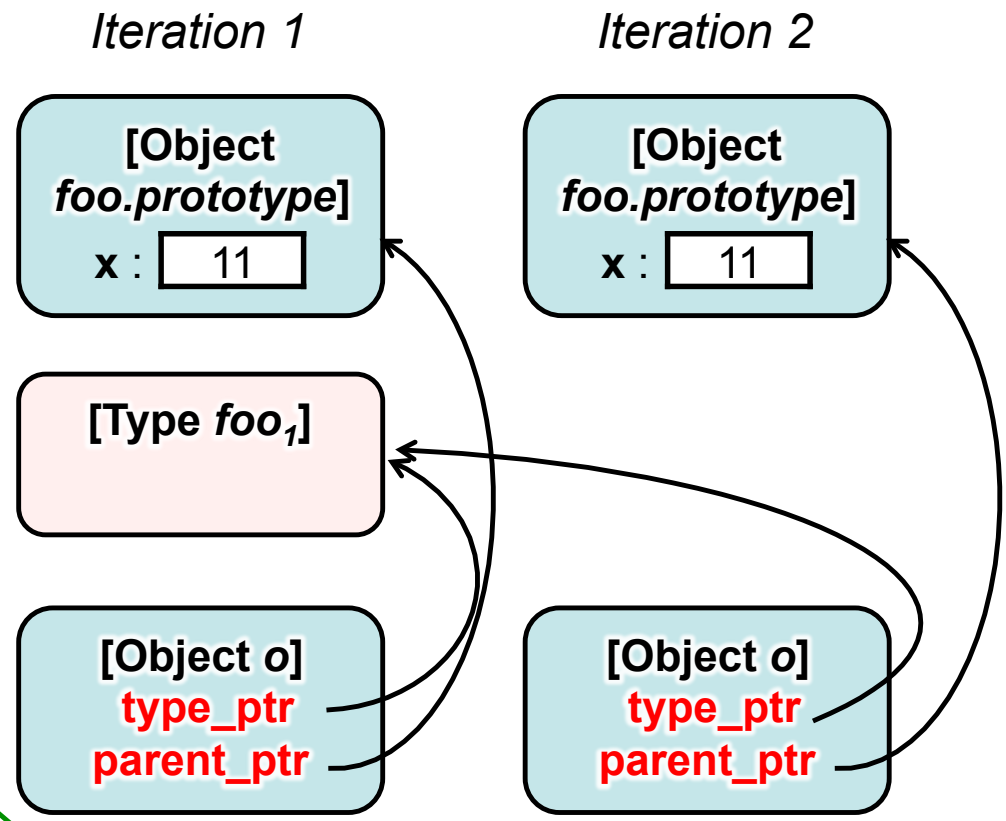
Solution: Decouple Inheritance from Types

```
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 */ }
```



IC Hit on *foo*$_1$
Repeated on all future iterations

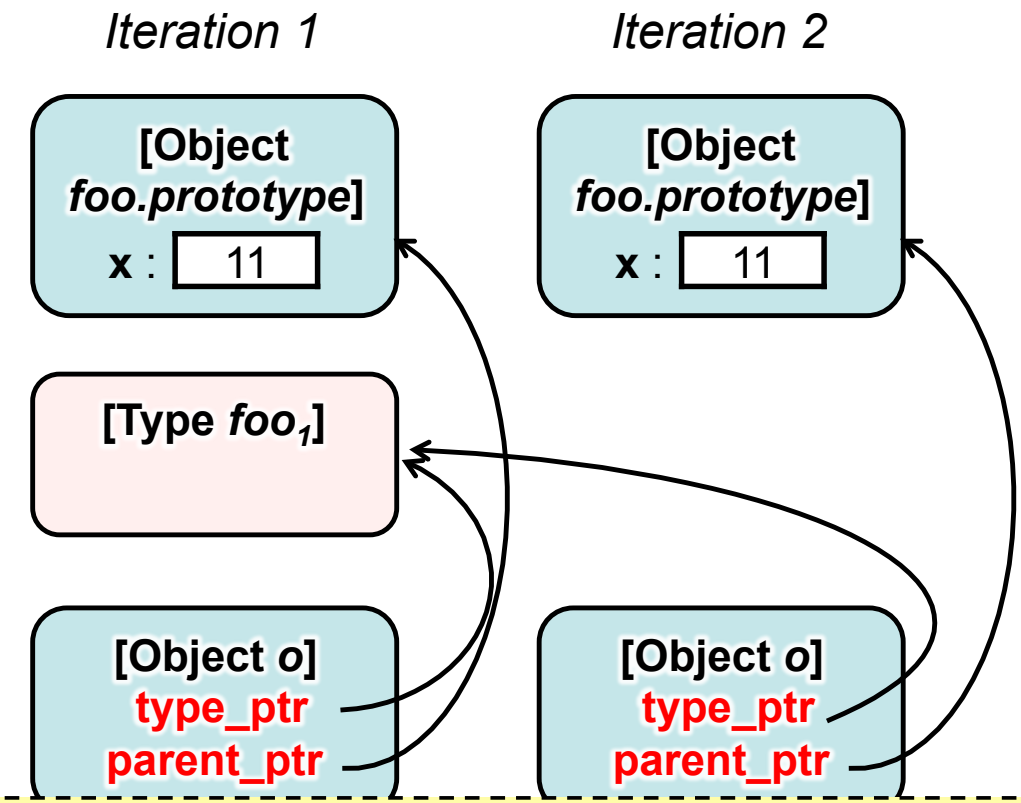
Solution: Decouple Inheritance from Types

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

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

[Inline Cache for *load o.x*]

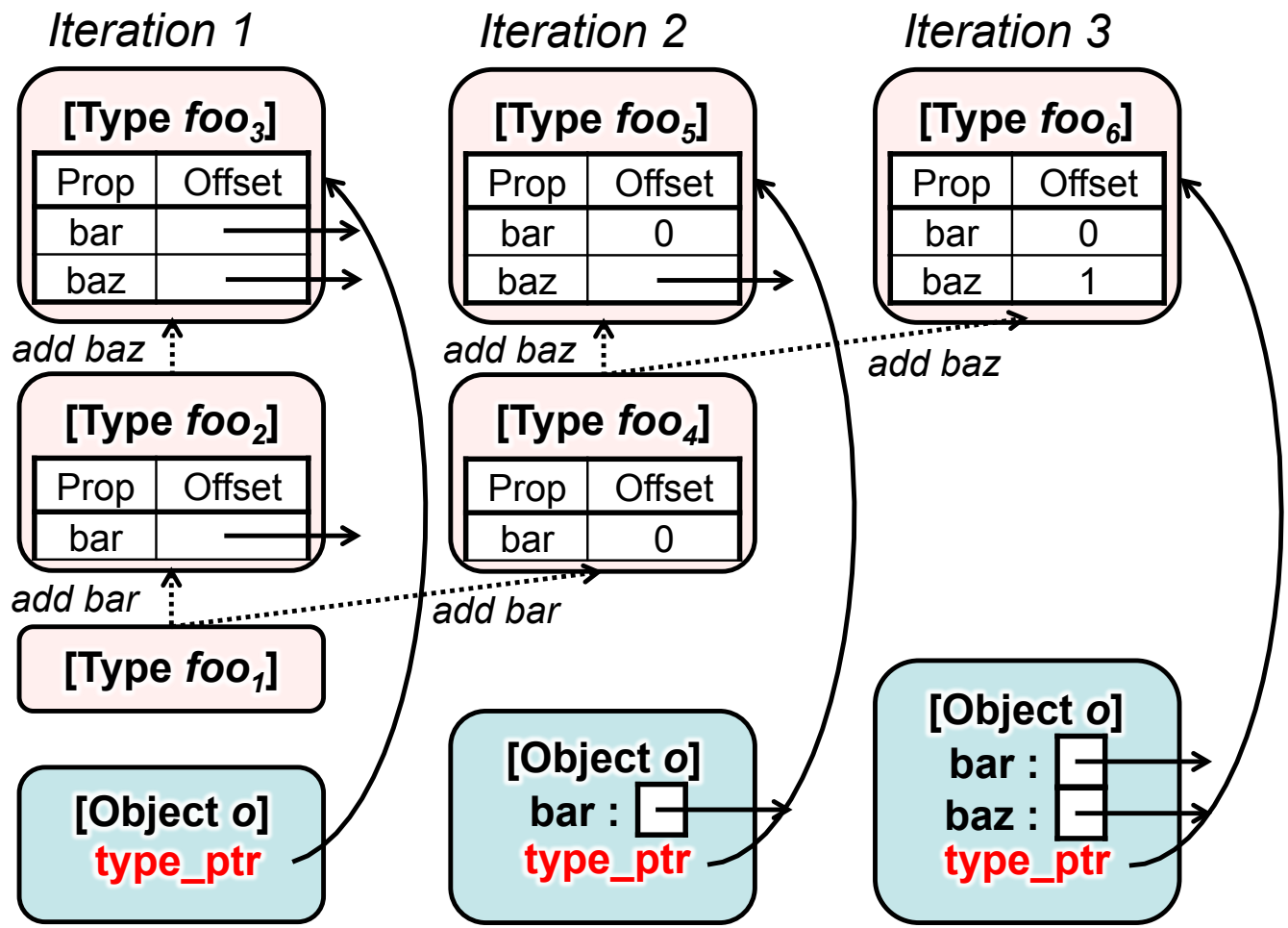
```
If (o.type_ptr == foo1) {
  prototype = o.proto_ptr;
```



- Together with method binding decoupling, removes almost all inline cache misses and most type polymorphism

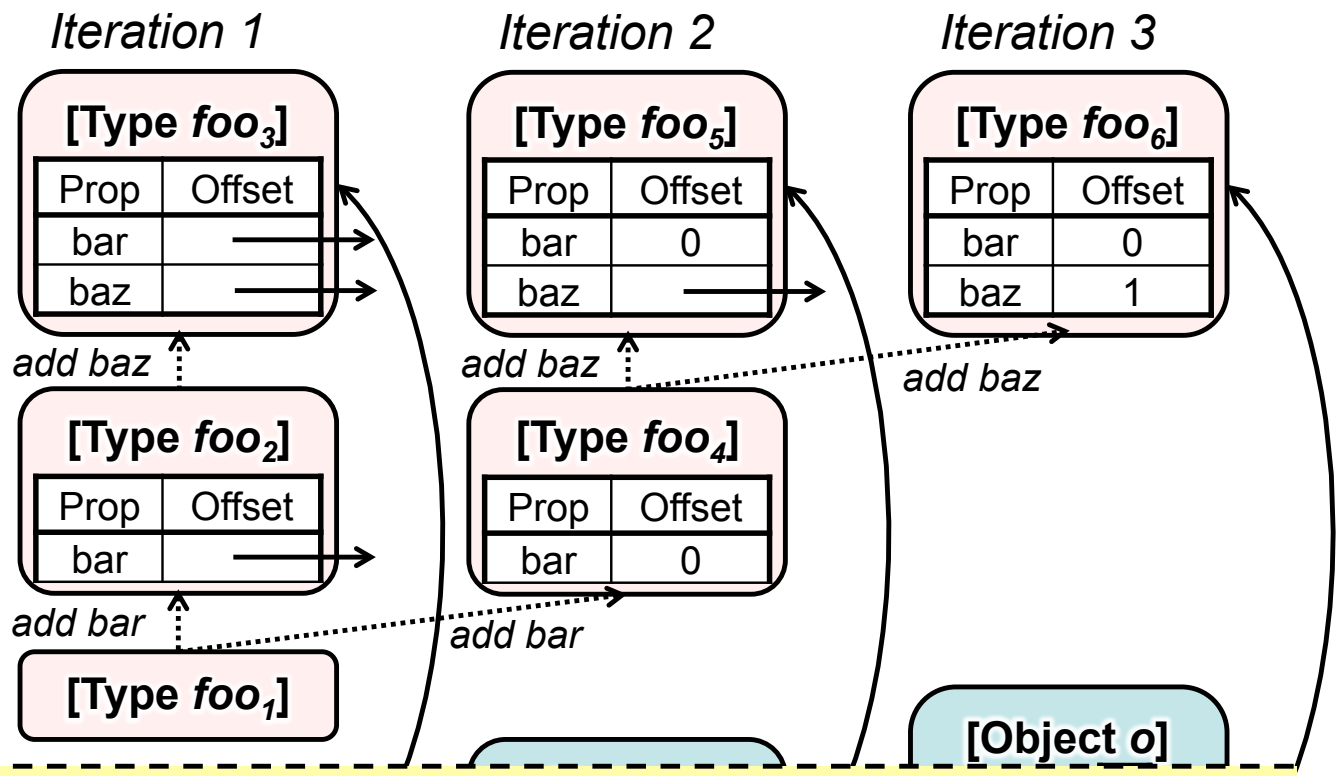
Problem: Rampant Type Creation Due to Encoding Method Bindings

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



Problem: Rampant Type Creation Due to Encoding Method Bindings

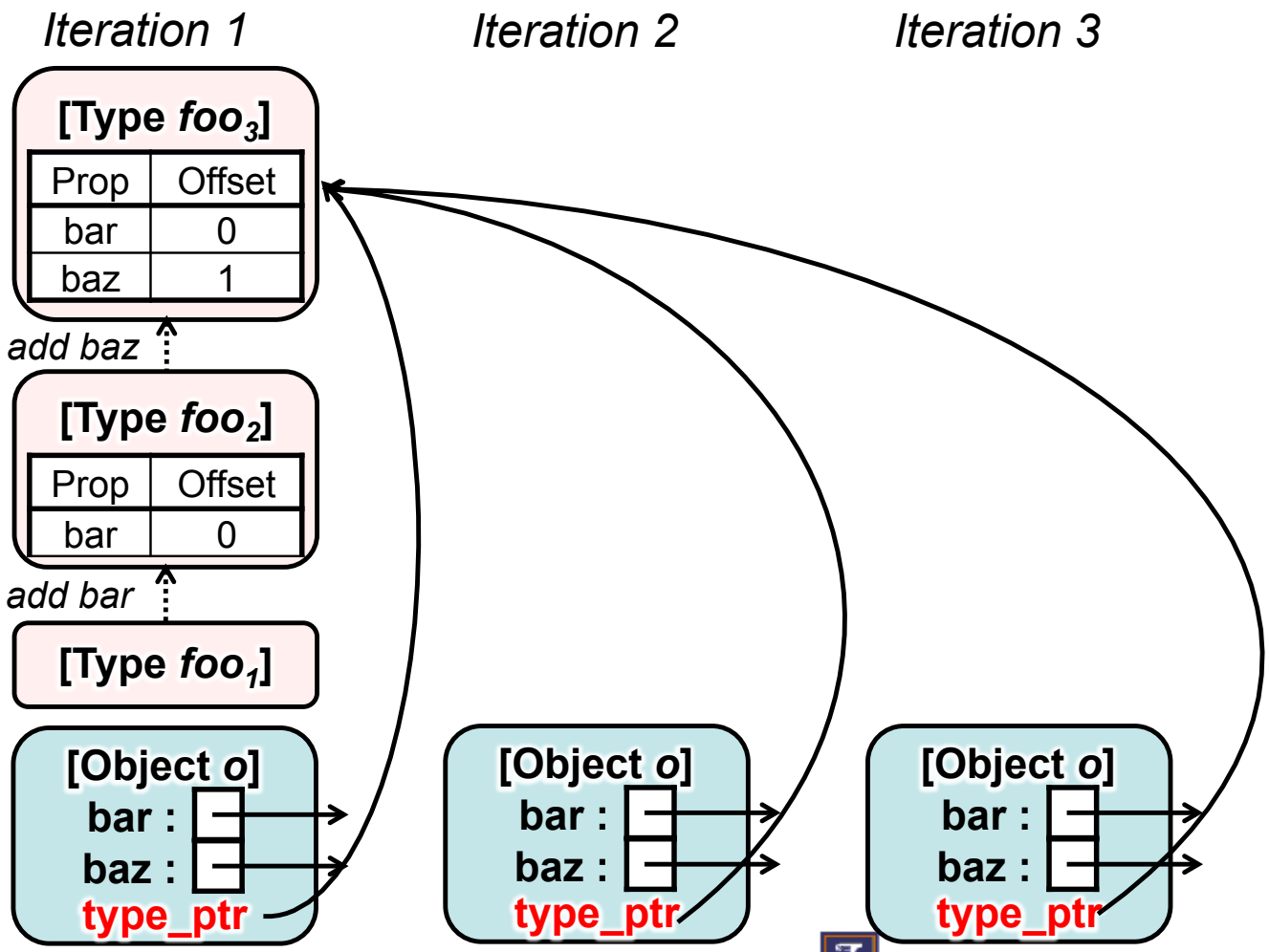
```
var foo = function() {
  this.bar = function() {
    ...
  };
  this.baz = function() {
    ...
  };
}
while (...) {
  var o = new foo();
}
```



- 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

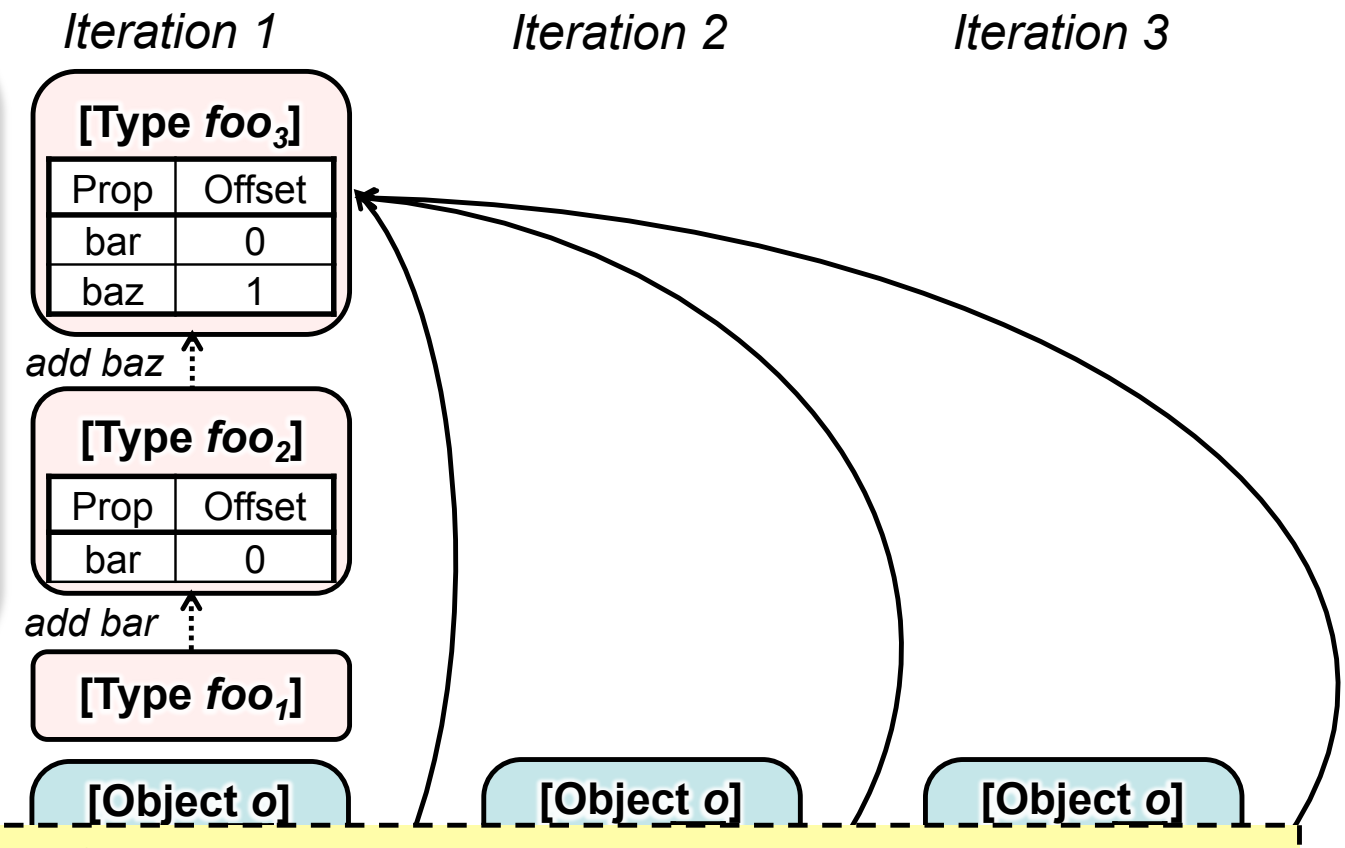
Solution: Decouple Method Bindings from Types

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



Solution: Decouple Method Bindings from Types

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

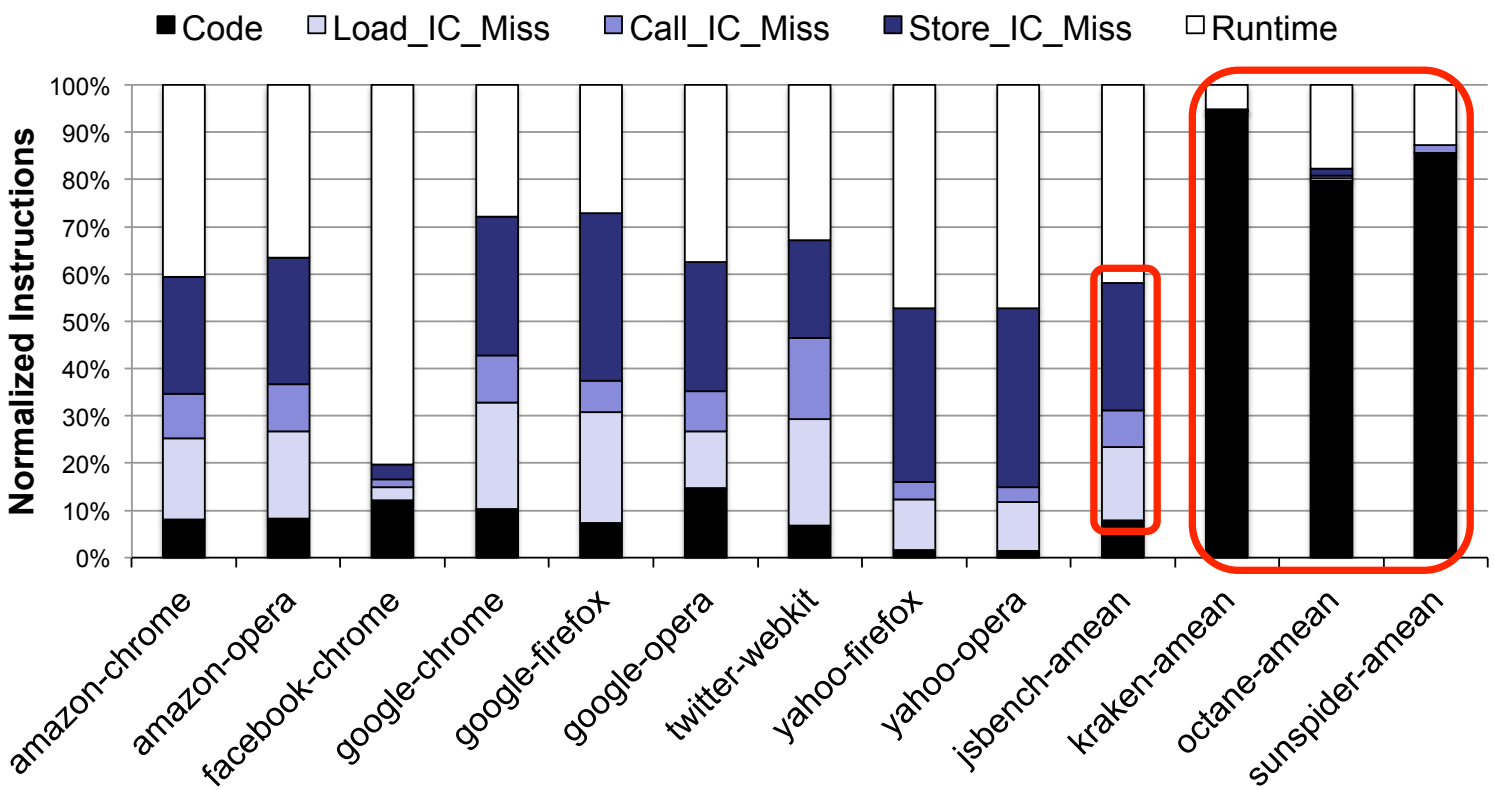


- Same type at end of every iteration
- O(N) type objects created in total (vs. $O(N^2)$ before)

Experimental Setup

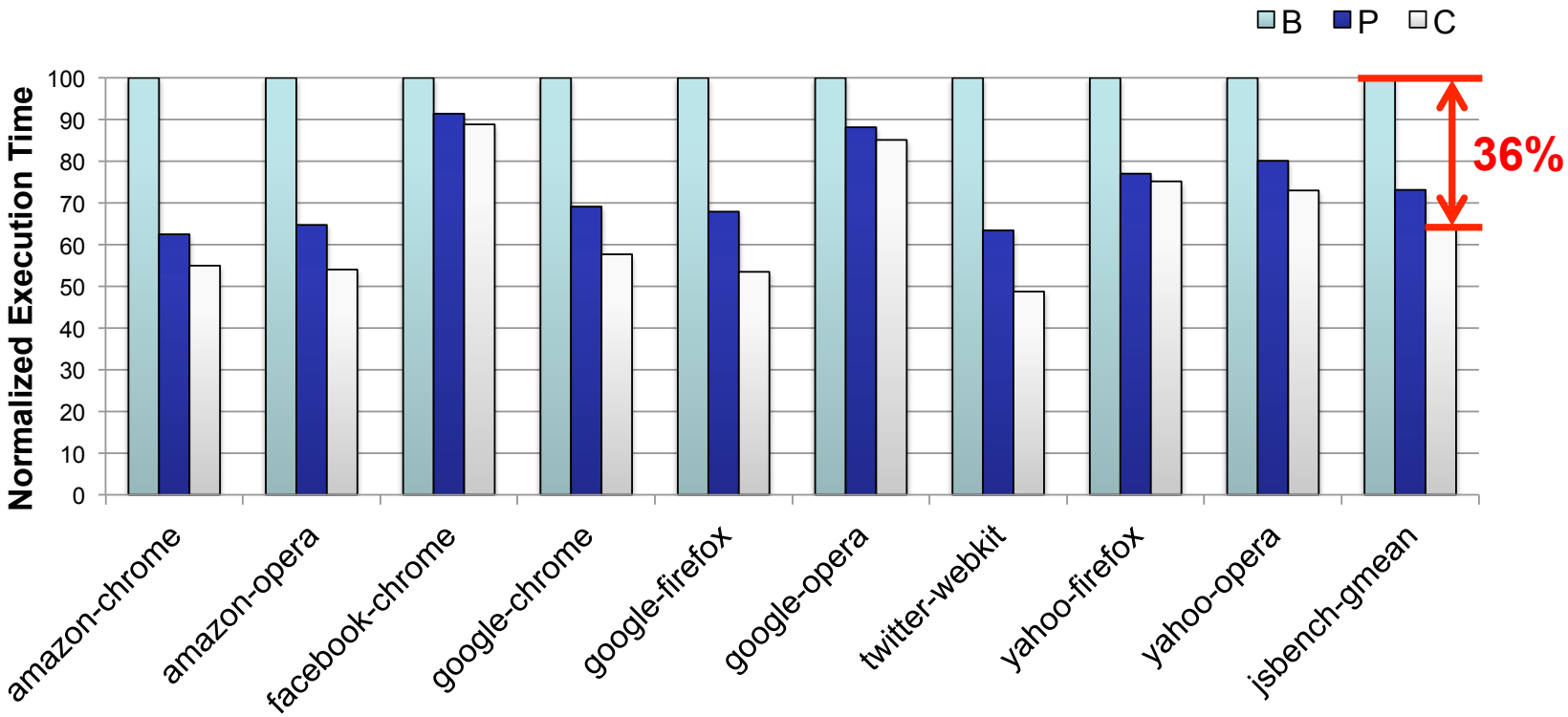
- Modified V8 Full Compiler with type decoupling enhancement
- Benchmark Suites: JSBench (websites), Kraken, Octane, Sunspider
  - 3 warm-ups before taking measurements to get steady state
- Measurements
  - Xeon machine (performance), Pin tool (characterization)
- Tested three configurations:
  - ***B***: Baseline Chrome V8
  - ***P***: Prototype inheritance decoupling
  - ***C***: Combined inheritance & method binding decoupling

Instruction Breakdown for Chrome V8



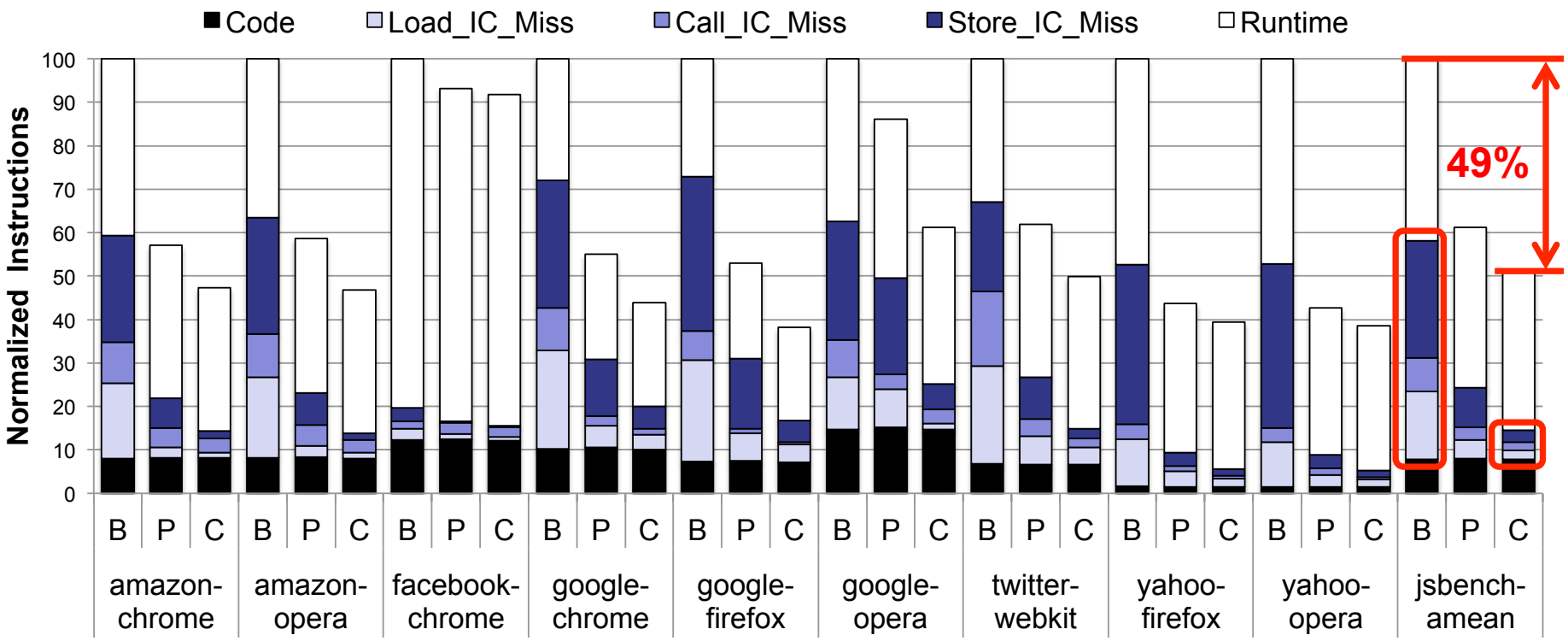
- JSBench: 50% of instructions spent on inline cache miss handling
- Kraken, Octane, Sunspider: Most instructions spent on generated code

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

i-acoma group

ILLINOIS