V8

by @dimko1

What?

- Googles Open Source Engine
- Written on C++
- Compiles JS into Machine Code at execution (JIT) without producing byte code
- Powers Chrome, Node, Opera
- Can run standalone, or can be embedded into any C++ program


Made in Germany
LBC9 HD WALLPAPERS


FASTEST
%ENGINE NAME%
risovach.ru

So, why to know?

- Have no idea:)

Demo

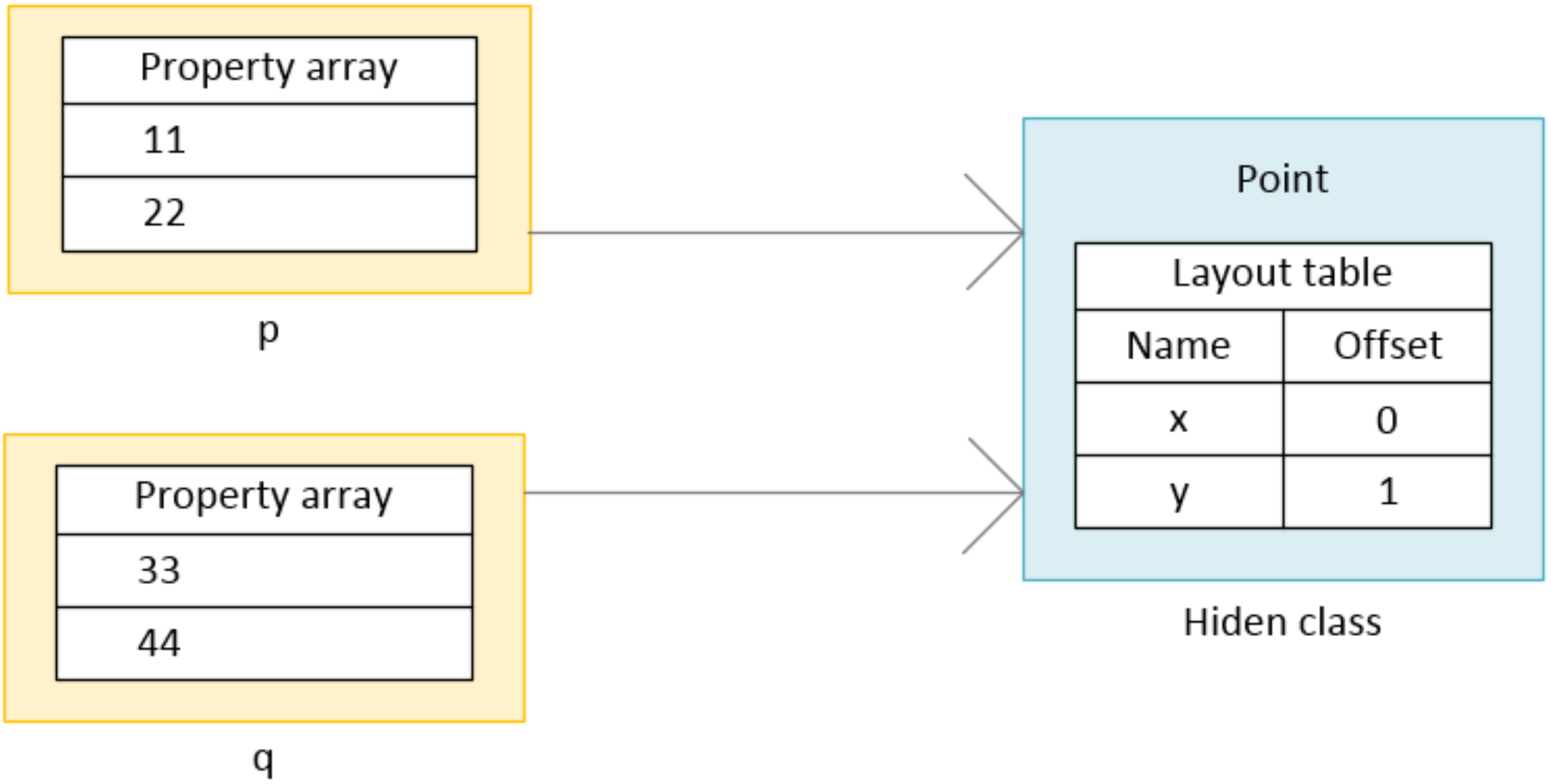
Hidden Classes

- We don’t have types. (oh c’mon, not about primitives)
- To optimise we need types…
- Types information is valuable for code generation
- Remember: Compilation during Execution

Hidden Classes help to run faster

- Creating hidden classes for objects during run-time
- Objects with same hidden class can use same optimised code

```
function Point(x, y) {
        this.x = x;
        this.y = y;
}
var p = new Point(11,22);
var q = new Point(33,44);
```



```
function Point(x, y) {
        this.x = x; // Point_1
        this.y = y; // Point_2
}
var p = new Point(11,22); // Point_0
var q = new Point(33,44);
q.z = 55; // Point_3
```

Property array
11
22

p

Property array
33
44
55

q

Point_0

Layout table	
Name	Offset

Point_1

Layout table	
Name	Offset
x	0

Point_2

Layout table	
Name	Offset
x	0
y	1

Point_3

Layout table	
Name	Offset
x	0
y	1
z	2

Summary

- Initialise all members in construction function
- Initialise members in same order

Tagging

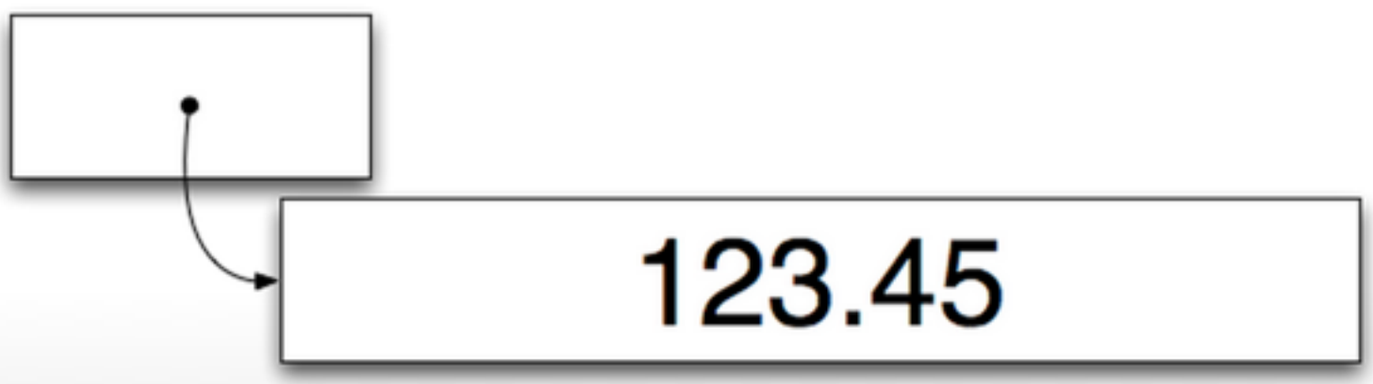
- V8 represent JavaScript objects with 32 bits values
- Object has flag 1
- Integer has flag 0 and called SMI
- If bigger - turning it into double and create new object

V8 uses **tagging**

- Objects

- **SM**all **I**ntegers

- Boxed double



Summary

- Prefer numeric values that can be represented as 31-bit integer

Arrays

- We have arrays, huge array and sparse arrays
- Two ways of representing arrays:
  - Fast Elements
  - Dictionary Elements

Summary

- Create arrays from 0 index :)
- Don’t pre-allocate large Arrays
- Don’t delete element from array
- Don't load uninitialized or deleted elements

JAVASCRIPT

```
a = new Array();
for (var b = 0; b < 10; b++) {
  a[0] |= b;  // Oh no!
}
```

vs.

JAVASCRIPT

```
a = new Array();
a[0] = 0;
for (var b = 0; b < 10; b++) {
  a[0] |= b;  // Much better! 2x faster.
}
```

JAVASCRIPT

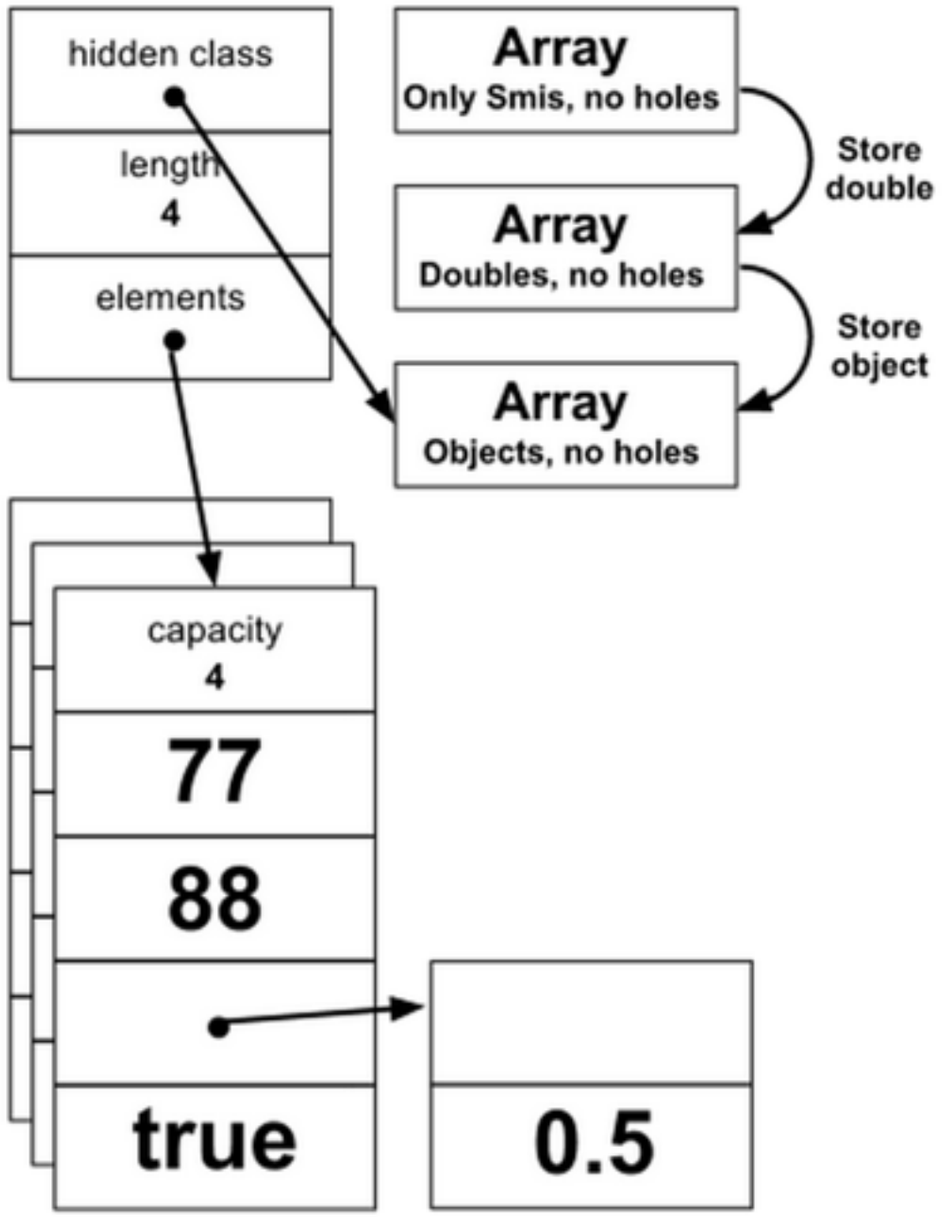
```
var a = new Array();

a[0] = 77;    // Allocates
a[1] = 88;
a[2] = 0.5;   // Allocates, converts
a[3] = true; // Allocates, converts
```

JAVASCRIPT

```
var a = new Array();

a[0] = 77;   // Allocates
a[1] = 88;
a[2] = 0.5;   // Allocates, converts
a[3] = true; // Allocates, converts
```



Summary 2

- User Array Literal: var a = [77, 88, 0.5, true]
- Don’t store non numeric values in numeric arrays

Compilers

- "Full" compiler can generate good code for any JavaScript
- Optimizing compiler produces great code for most JavaScript

Full Compiler

- Generate code quickly
- Does do no type analysis
- Using Inline Caching. Gather type information in runtime

How Inline Cache Works

- Type dependant code for operations
- Validate type assumptions first
- Change at runtime as more types discovered

```
candidate % this.primes[i]
```

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call LoadIC_Initialize          ;; this.primes
push eax
mov eax,[ebp+0xf4]
pop edx
mov ecx,eax
call KeyedLoadIC_Initialize     ;; this.primes[i]
pop edx
call BinaryOpIC_Initialize Mod  ;; candidate % this.primes[i]
```

IA32 ASSEMBLY

```
...
push [ebp+0x8]
mov eax,[ebp+0xc]
mov edx,eax
mov ecx,0x50b155dd
call 0x311286e0
push eax
mov eax,[ebp+0xf4]
pop edx
mov ecx,eax
call 0x31129ae0
pop edx
call 0x3112ade0
...
```

LOAD IC

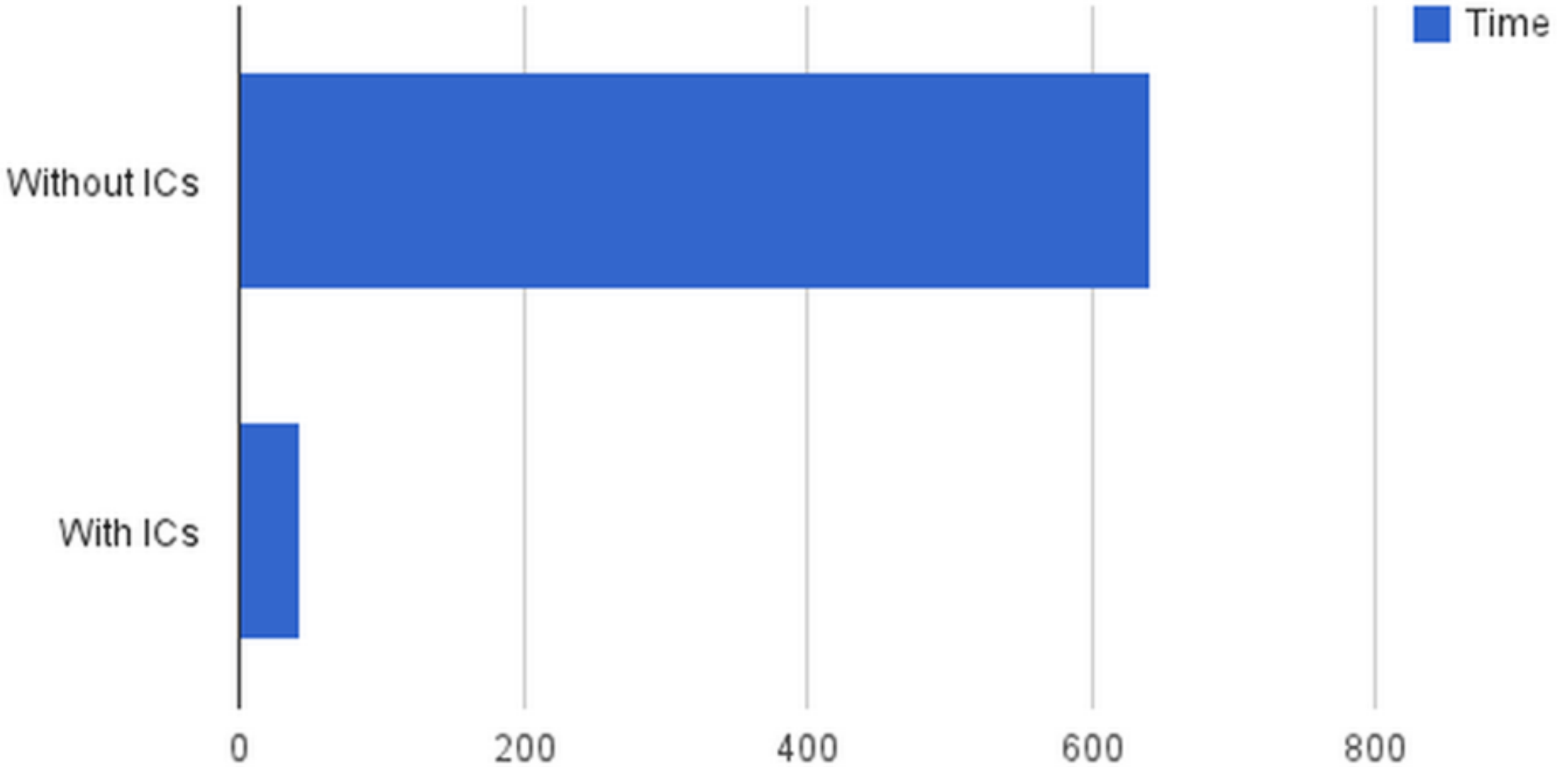
```
;; Code that knows how to
;; get fetch primes from a Prime object
...
ret
```

KEYED LOAD IC

```
;; Code that knows
;; how to get an element from SMI Array
...
ret
```

BINARY OP IC

```
;; Code that knows
;; how to calculate SMI % SMI
...
ret
```


Time
Without ICs
With ICs
0
200
400
600
800

Monomorphic better than Polymorphic

JAVASCRIPT

```
function add(x, y) {
  return x + y;
}

add(1, 2);       // + in add is monomorphic
add("a", "b");   // + in add becomes polymorphic
```

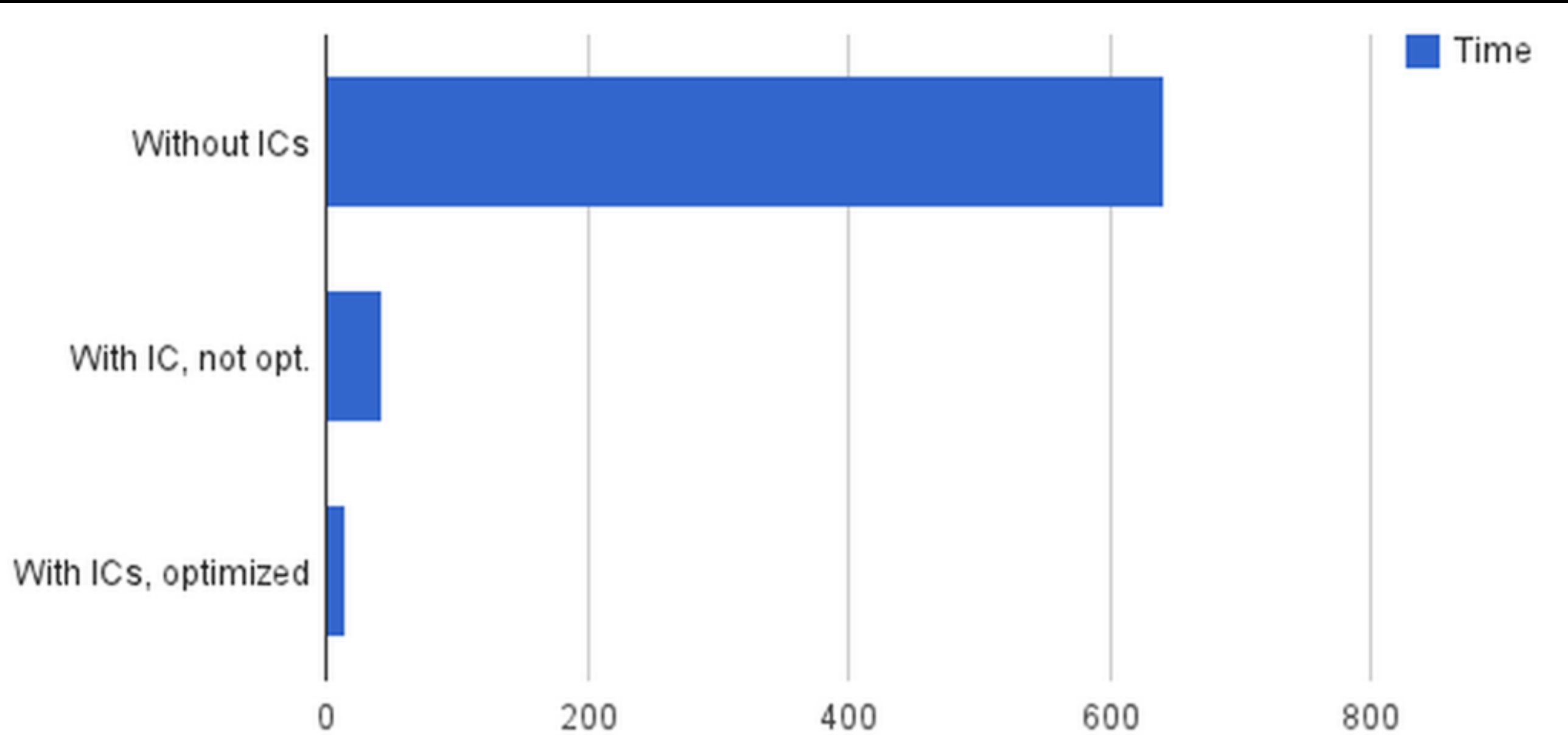
Optimizing Compiler

- Comes later and re-compiles “hot” functions
- Types taken form ICs
- Monomorphic can be inlined
- Inlining enables other optimizations

`candidate % this.primes[i]`

IA32 ASSEMBLY

```
cmp [edi+0xff],0x4920d181   ;; Is this a Primes object?
jnz 0x2a90a03c
mov eax,[edi+0xf]           ;; Fetch this.primes
test eax,0x1                ;; Is primes a SMI ?
jz 0x2a90a050
cmp [eax+0xff],0x4920b001   ;; Is primes hidden class a packed SMI array?
mov ebx,[eax+0x7]
mov esi,[eax+0xb]           ;; Load array length
sar esi,1                   ;; Convert SMI length to int32
cmp ecx,esi                 ;; Check array bounds
jnc 0x2a90a06e
mov esi,[ebx+ecx*4+0x7]     ;; Load element
sar esi,1                   ;; Convert SMI element to int32
test esi,esi                ;; mod (int32)
jz 0x2a90a078
...
cdq
idiv esi
```


Time
Without ICs
With IC, not opt.
With ICs, optimized
0
200
400
600
800

Deoptimization

- Optimization are speculative
- Throws away optimized code
- Resumes execution at the right place
- Reoptimization might be triggered again later

Summary

- Avoid changes in the hidden classes after functions were optimised

What is a problem now?

- Ensure problem is in JS
- Reduce to pure JS (not DOM)
- Collect metrics

Demo

?