Google™ 09 I O

V8 Internals

Mads Sig Ager
May 27, 2009

Agenda

- Why a new JavaScript engine?
- V8 design overview
- V8 internals
  - Hidden classes
  - Inline caching
  - Precise generational garbage collection
- Recent developments
  - Irregexp: new JS regular expression engine
  - New compiler infrastructure
- Object heap scalability
- Performance bottlenecks

Why a New JavaScript Engine?

Why Build a New JavaScript Engine?

- When the development of V8 started, the existing JavaScript engines were slow
  - Interpreters operating on AST or bytecodes
  - Poor memory management with big GC pauses
- High performance JavaScript engines are key to continued innovation for web applications
- Starting from scratch seemed like the best approach
- The goal is to push the performance bar for JavaScript

The Challenge

- JavaScript is a highly dynamic language
- Objects are basically hash maps
- Properties are added and removed on-the-fly
- Prototype chains are modified during execution
- 'eval' can change the calling context
- 'with' introduces objects in the scope chain dynamically

V8 Design Decisions

Design Goals

- Make large object-oriented programs perform well
- Fast property access
- Fast function calls
- Fast and scalable memory mangement

Key V8 Components

- Hidden classes and class transitions
- Compilation to native code with inline caching
- Efficient generational garbage collection

V8 Internals

V8 Memory Model

- 32-bit tagged pointers
- Objects are 4-byte aligned, so two bits available for tagging
- Small 31-bit signed integers are immediate values distinguished from pointers by tags

Small integer	XXXX...XXXX0
Pointer	XXXX...XXX01

- Base JavaScript objects consists of three words

Hidden Class Pointer
Properties Pointer
Elements Pointer

Hidden Classes

- Wanted to take advantage of optimization techniques from statically typed object oriented languages
- Introduced the concept of hidden classes to get there
- Hidden classes group objects that have the same structure

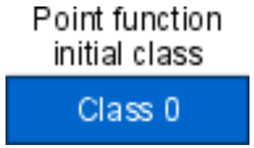
Hidden Classes by Example

- JavaScript objects constructed in the same way should get the same hidden class

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```

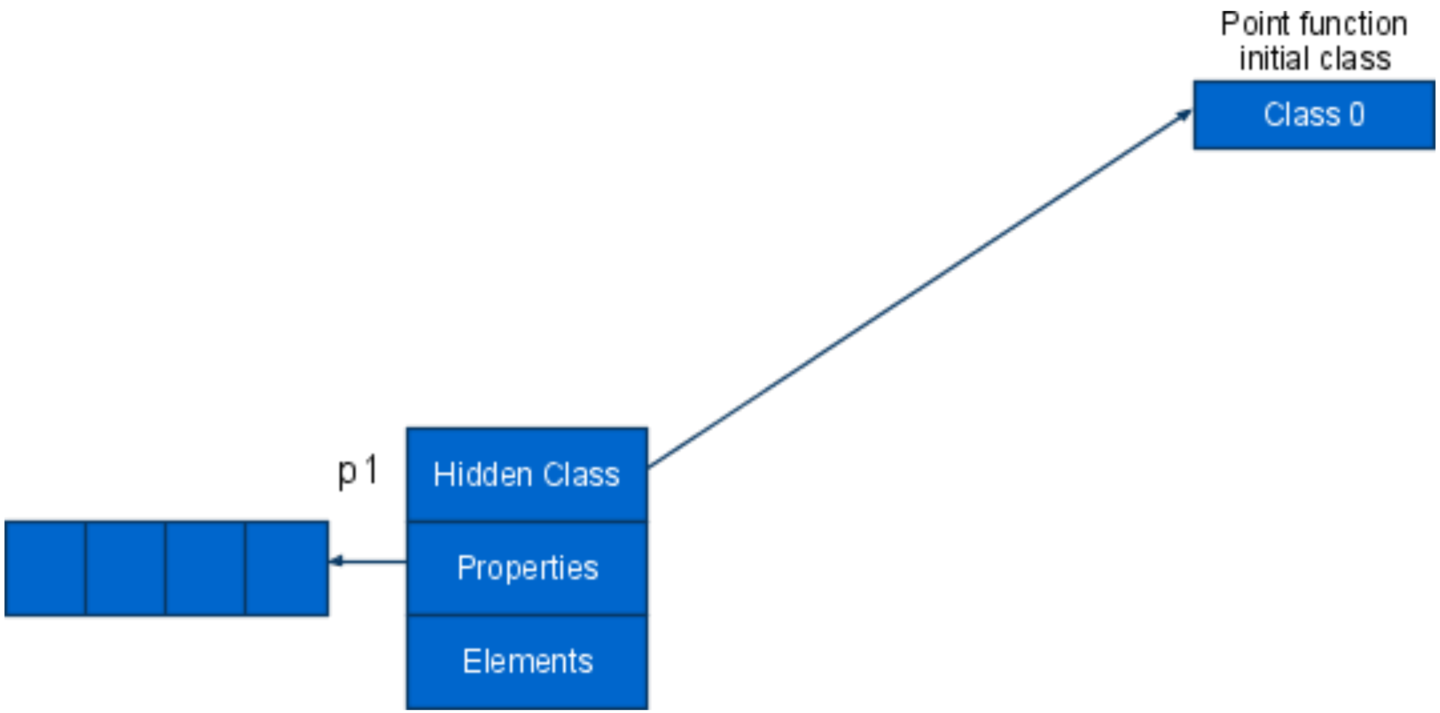
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



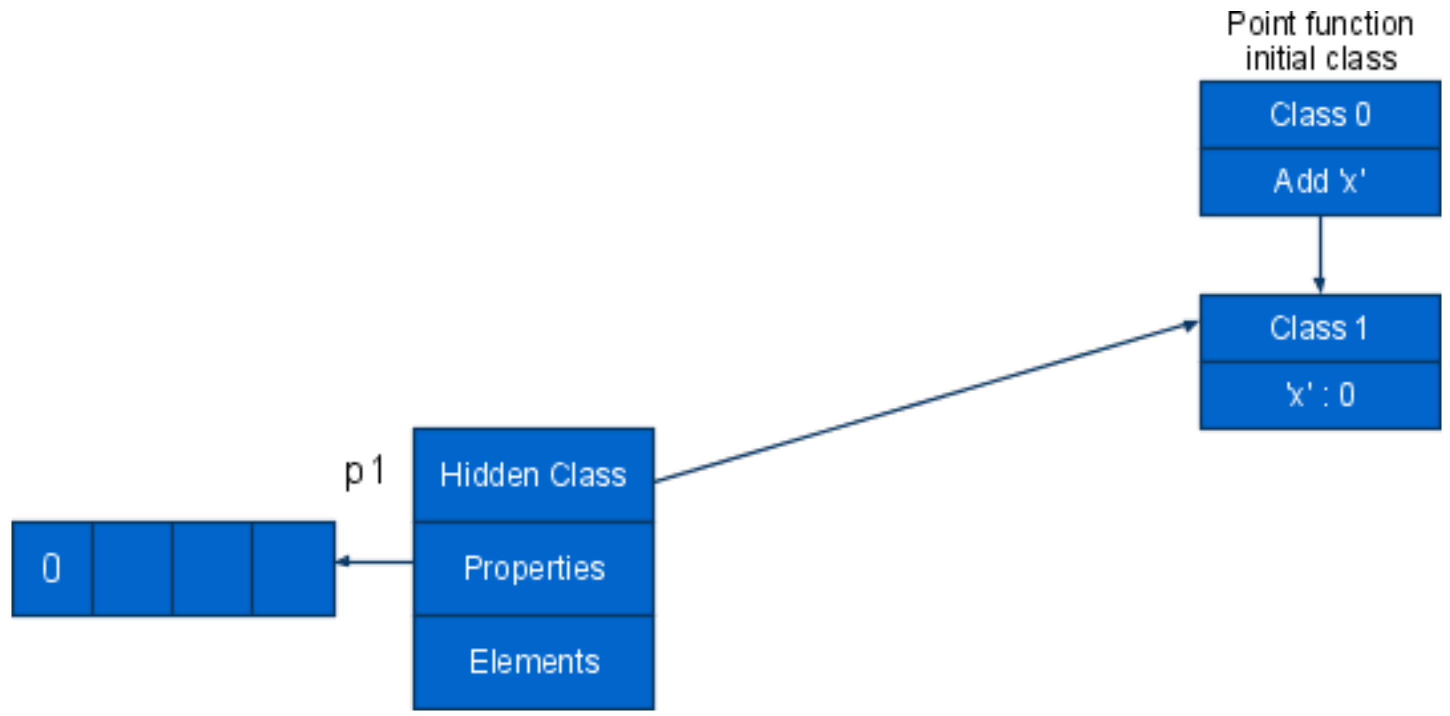
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



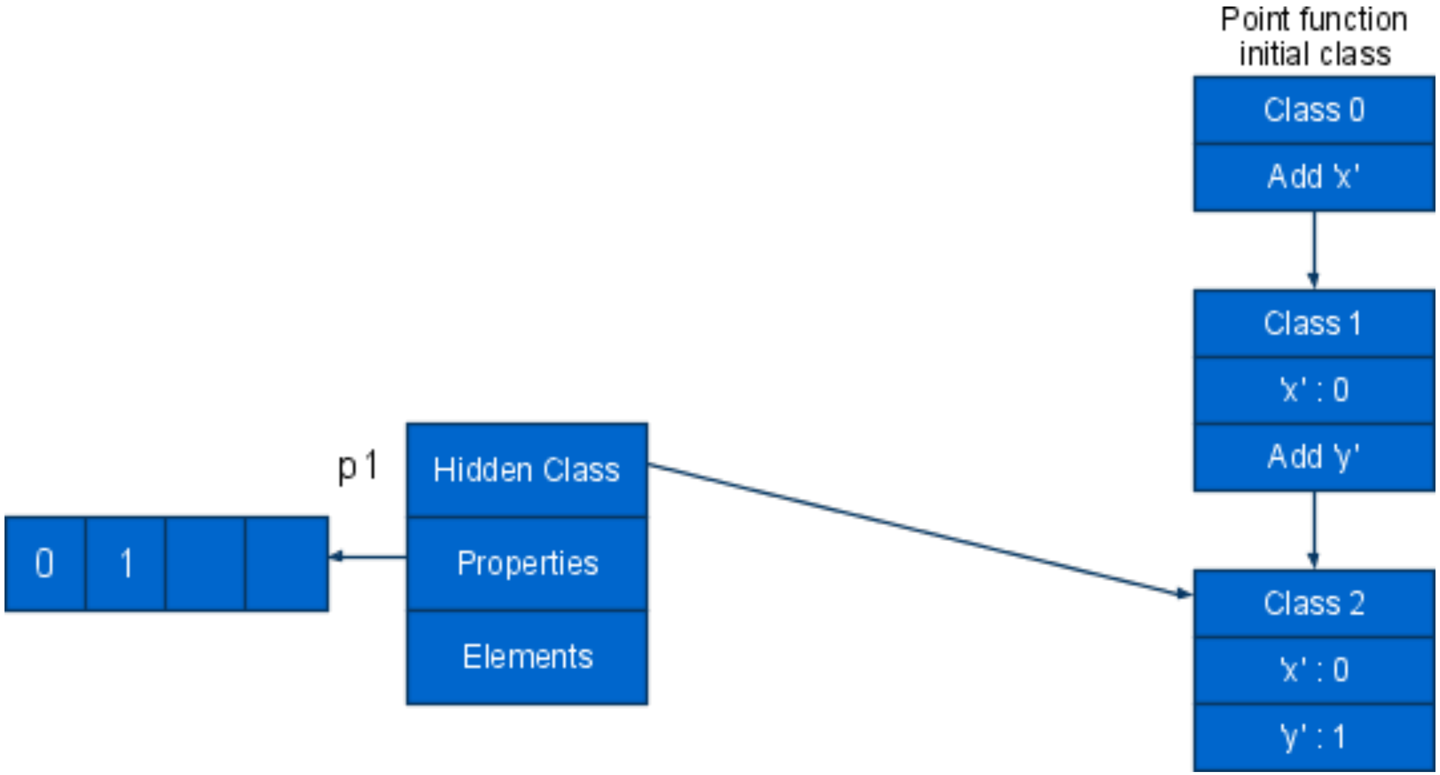
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



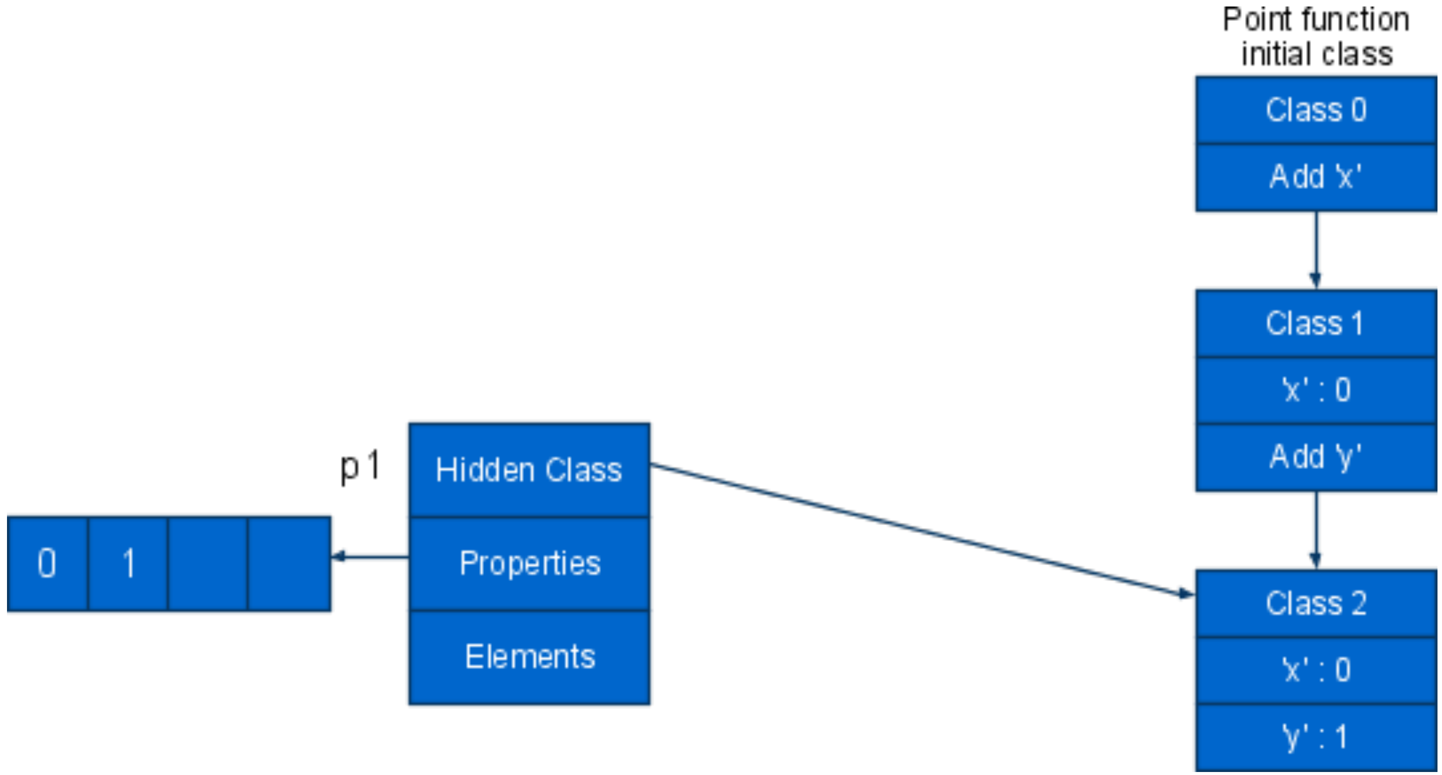
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



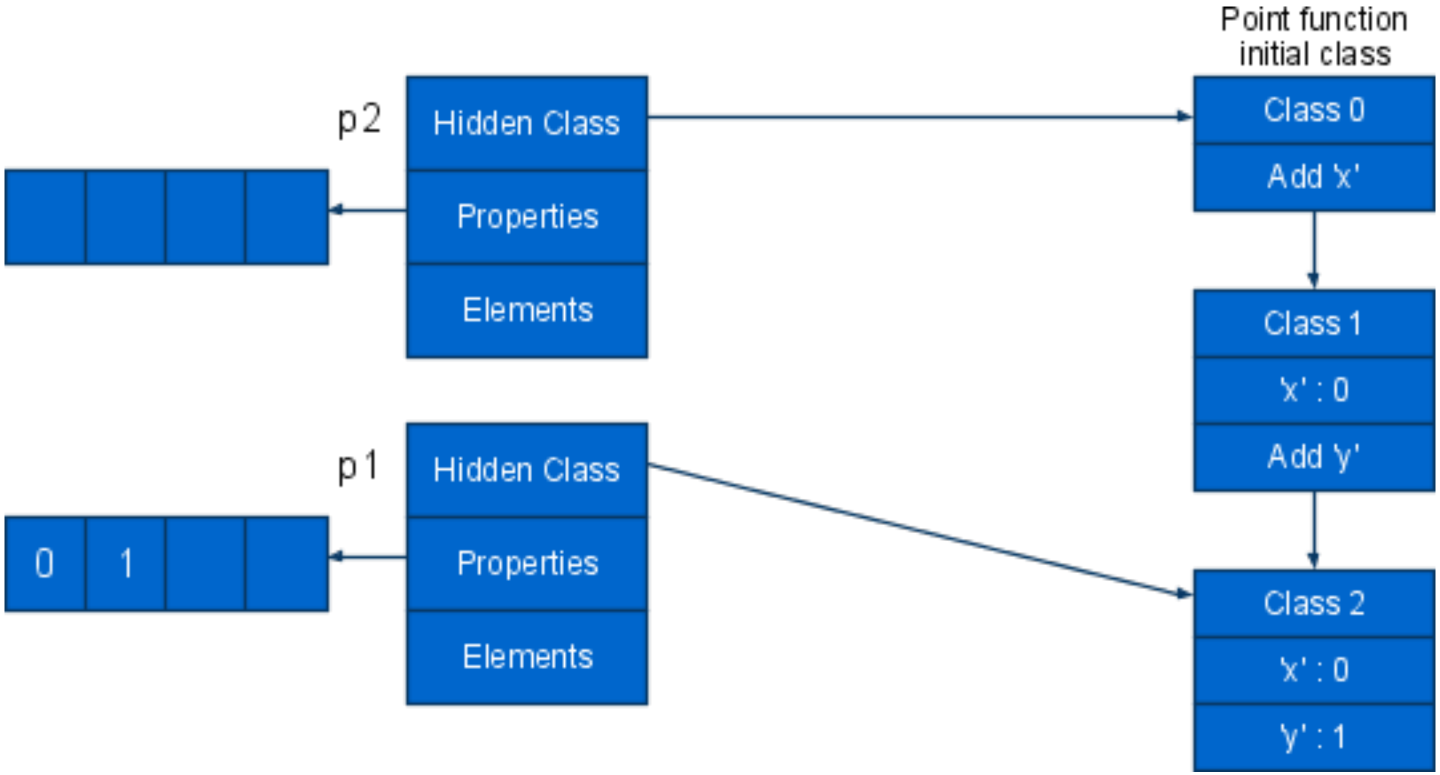
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



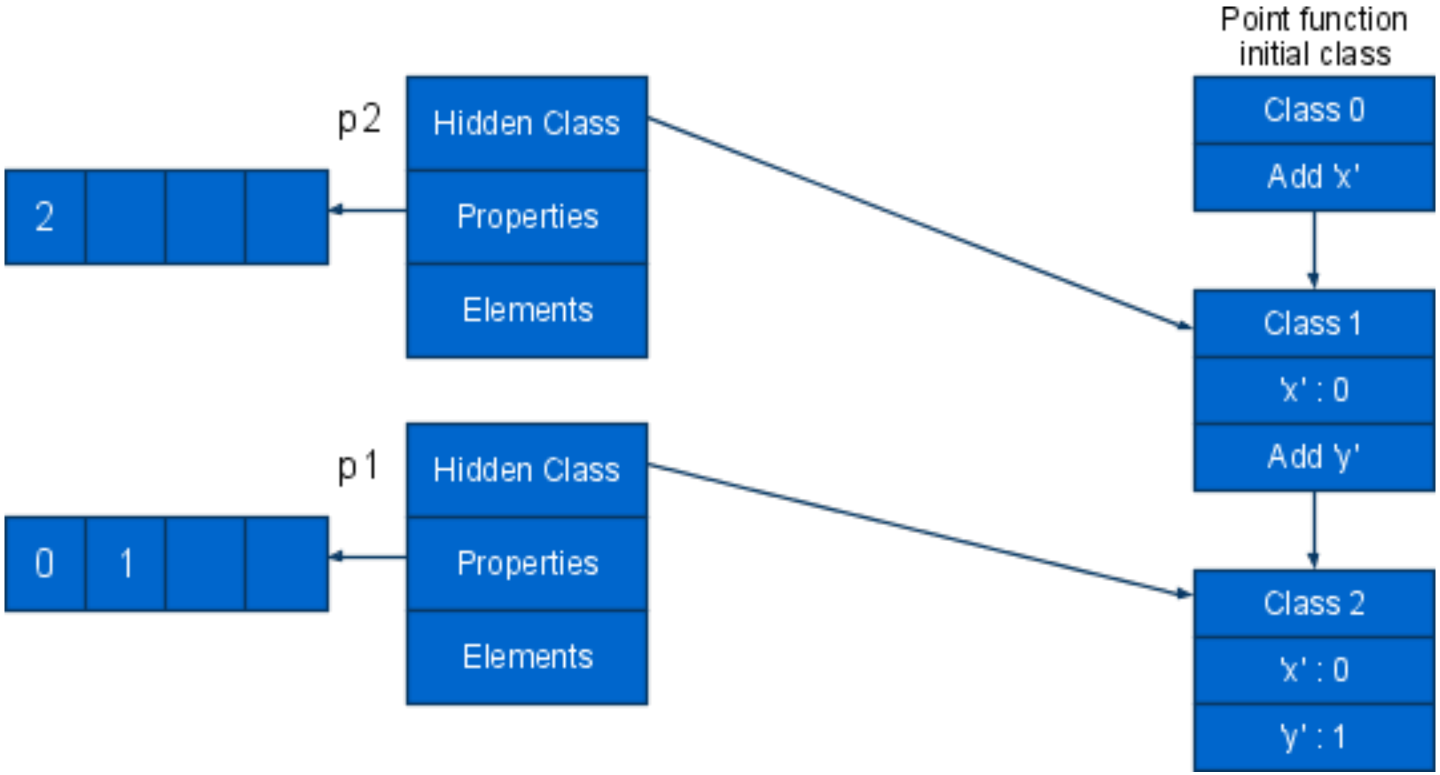
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



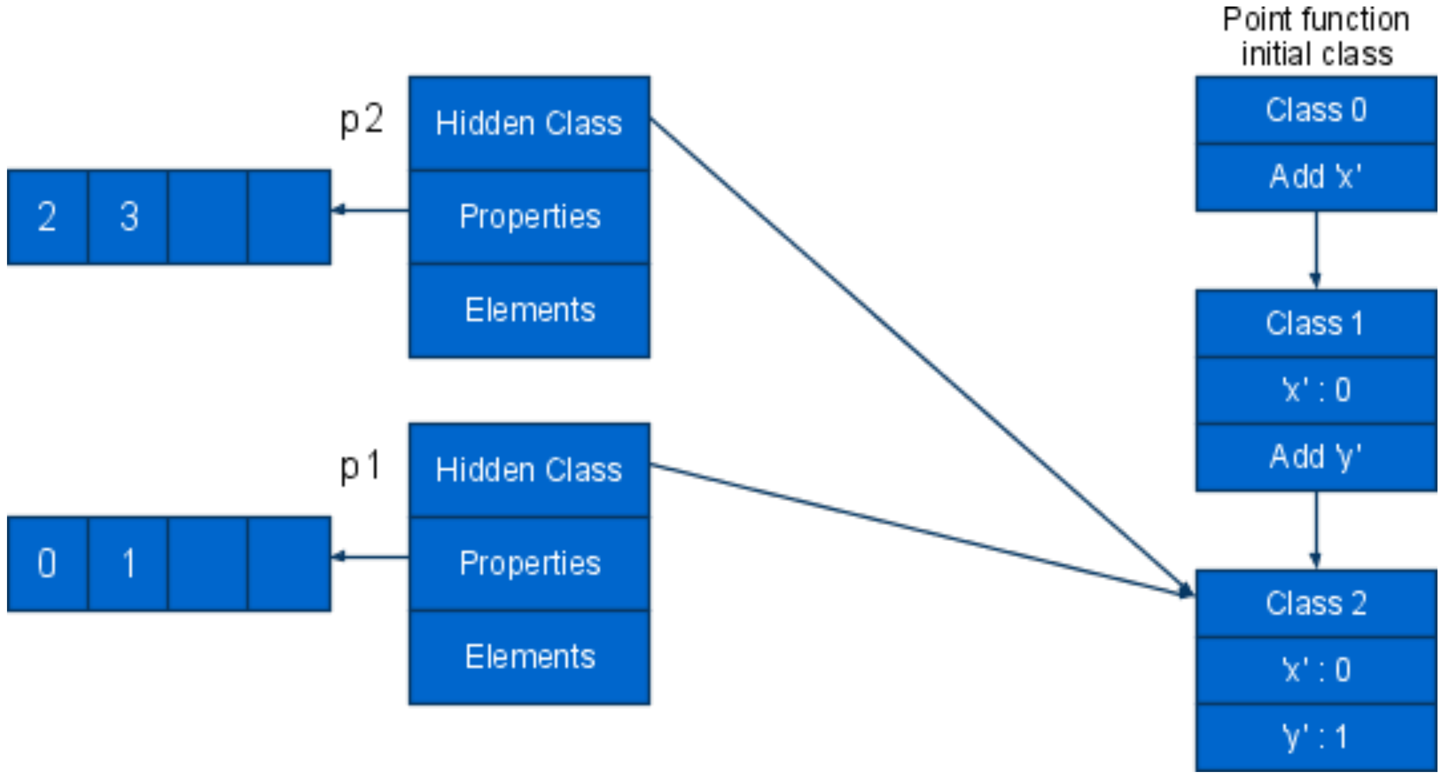
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



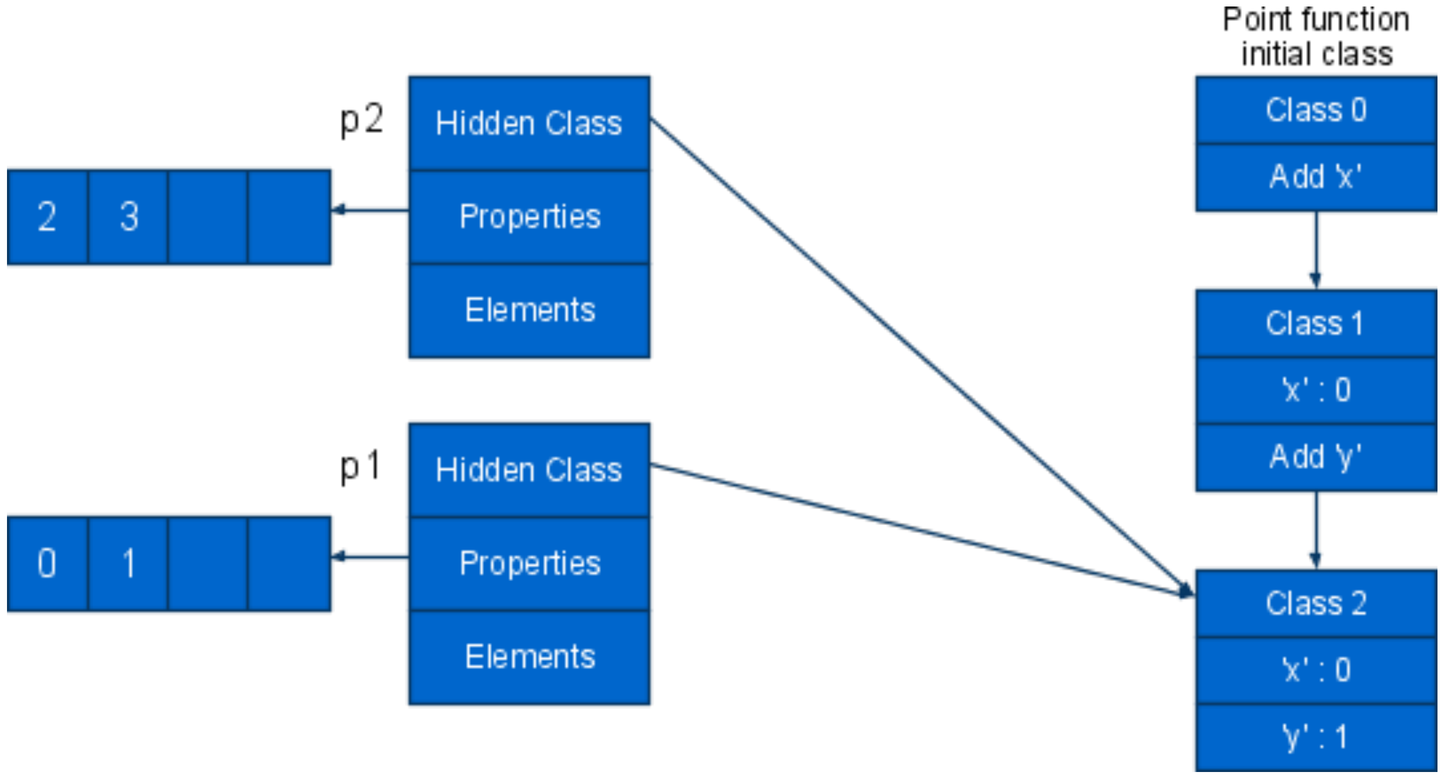
Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



Hidden Classes by Example

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
var p1 = new Point(0,1);
var p2 = new Point(2,3);
```



How Dynamic is JavaScript?

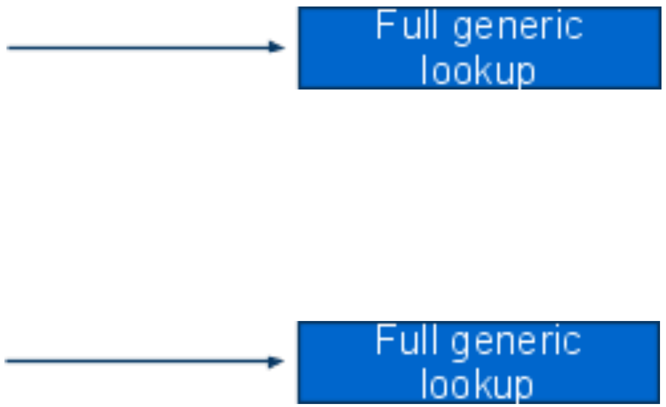
- In 90% of the cases, only objects having the same map are seen at an access site
- Hidden classes provides enough structure to use optimization techniques from more static object-oriented language
- We do not know the hidden class of objects at compile time
- We use runtime code generation and a technique called inline caching

Inline Caching

...

load 'x'

load 'y'



...

Inline Caching

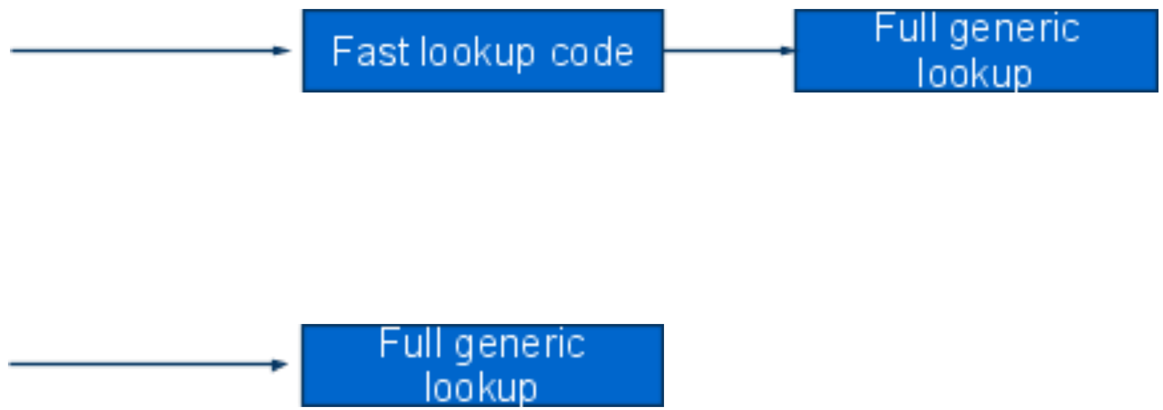
...

load 'x'

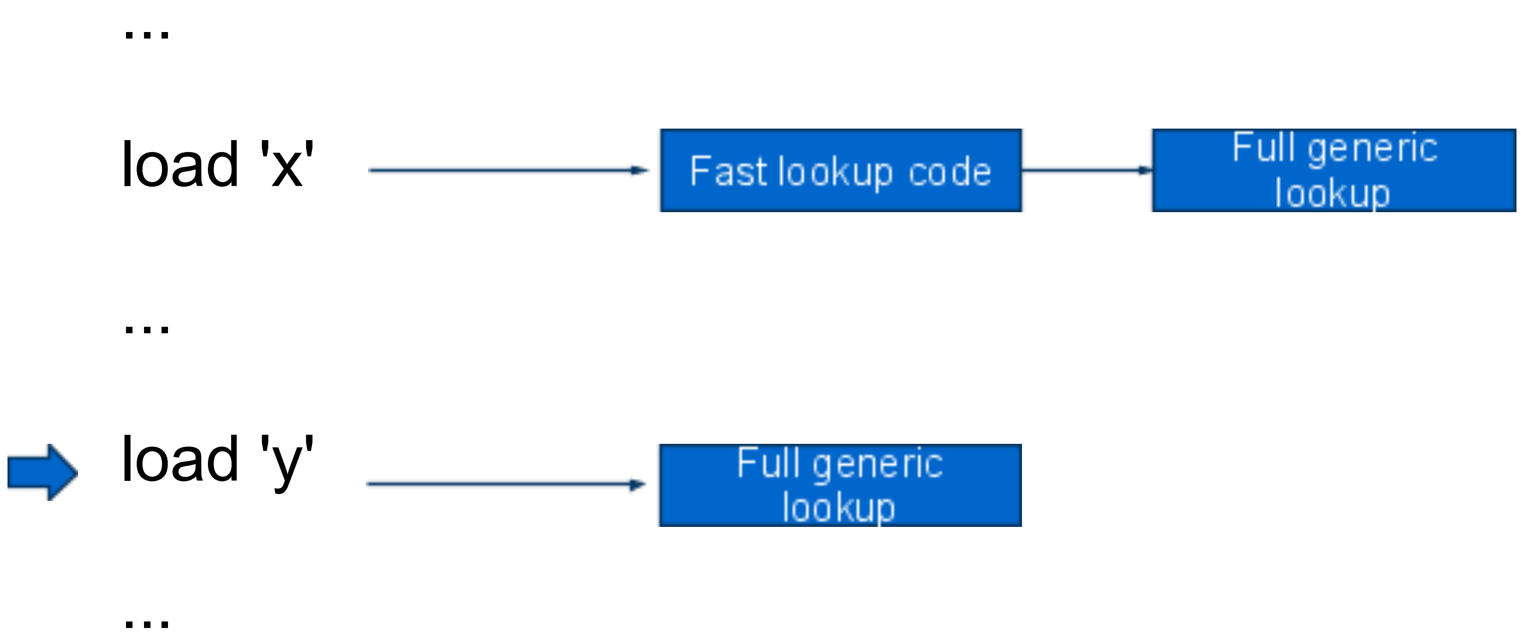
...

load 'y'

...



Inline Caching



Inline Caching

...

load 'x'

...

load 'y'

...

Monomorphic Load Inline Cache

```
0xf7c0d32d (size = 37):
 0 mov eax,[esp+0x4]
 4 test al,0x1
 6 jz 32
12 cmp [eax+0xff],0xf78fab81
19 jnz 32
25 mov ebx,[eax+0x3]
28 mov eax,[ebx+0x7]
31 ret
32 jmp LoadIC_Miss
```

Monomorphic Load Inline Cache

```
0xf7c0d32d (size = 37):
 0 mov eax,[esp+0x4]          ; receiver load
 4 test al,0x1                ; object check
 6 jz 32
12 cmp [eax+0xff],0xf78fab81
19 jnz 32
25 mov ebx,[eax+0x3]
28 mov eax,[ebx+0x7]
31 ret
32 jmp LoadIC_Miss
```

Monomorphic Load Inline Cache

```
0xf7c0d32d (size = 37):
 0 mov eax,[esp+0x4]        ; receiver load
 4 test al,0x1              ; object check
 6 jz 32
12 cmp [eax+0xff],0xf78fab81 ; class check
19 jnz 32
25 mov ebx,[eax+0x3]
28 mov eax,[ebx+0x7]
31 ret
32 jmp LoadIC_Miss
```

Monomorphic Load Inline Cache

```
0xf7c0d32d (size = 37):
 0 mov eax,[esp+0x4]        ; receiver load
 4 test al,0x1              ; object check
 6 jz 32
12 cmp [eax+0xff],0xf78fab81 ; class check
19 jnz 32
25 mov ebx,[eax+0x3]        ; load properties
28 mov eax,[ebx+0x7]        ; load property
31 ret
32 jmp LoadIC_Miss
```

Monomorphic Load Inline Cache

```
0xf7c0d32d (size = 37):
 0 mov eax,[esp+0x4]        ; receiver load
 4 test al,0x1              ; object check
 6 jz 32
12 cmp [eax+0xff],0xf78fab81 ; class check
19 jnz 32
25 mov ebx,[eax+0x3]        ; load properties
28 mov eax,[ebx+0x7]        ; load property
31 ret
32 jmp LoadIC_Miss           ; fallback to
                          ; generic lookup
```

Inline Cache States

- Three inline cache states
  - Uninitialized
  - Monomorphic
  - Megamorphic
- In the megamorphic state a cache of generated stubs is used
- Inline caches are cleared on full garbage collections
  - Allows us to get rid of unused code stubs
  - Gives all inline caches a new chance to hit the monomorphic case

Efficient Memory Management

- Precise generational garbage collection
- Two generations
  - Young generation is one small, contiguous space that is collected often
  - Old generation is divided into a number of spaces that are only occasionally collected
    - Code space (executable)
    - Map space (hidden classes)
    - Large object space (>8K)
    - Old data space (no pointers)
    - Old pointer space
- Objects are allocated in the young generation and moved to the old generation if they survive in the young generation

Types of Garbage Collection

- Scavenge
  - Copying collection of only the young generation
  - Pause times ~2ms
- Full non-compacting collection
  - Mark-Sweep collection of both generations
  - Free memory gets added to free lists
  - Might cause fragmentation
  - Pause times ~50ms
- Full compacting collection
  - Mark-Sweep-Compact collection of both generations
  - Pause times ~100ms

Recent developments

Irregexp: New Regular Expression Engine

- V8 initially used a library from WebKit called JSCRE
- JSCRE did not fit well with the string types in V8 and did not perform well
- Implemented Irregexp giving a 10x speedup on regular expression matching on benchmark programs
- Irregexp implements full JS regular expressions and there is no fallback to other libraries

Irregexp Internals

- Builds an automaton from the input regular expression
- Performs analysis and optimization on the automaton
- Generates native code
- Uses a number of tricks to avoid backtracking
- Reorders operations to perform the least expensive operations first

Irregexp Examples

- Use masks to search for common parts in alternatives first
- To search for

/Sun|Mon/

- First search for

/??n/

- Avoids a lot of backtracking

Irregexp Examples

- Will match up to four characters at a time on ASCII strings
- For

/foobar/

- It will search for

0x666f6f62 0x6172

Irregexp Examples

- Perform cheap operations first
- For

/([fF]oo[bB]ar)/

- Perform the following actions
  - Match **oo** and **ar** at positions 1 and 4
  - Match **[fF]**at position 0
  - Match **[bB]**at position 3
  - Perform capture

New Compiler Infrastructure

- Original compiler was very simple
- No static analysis of any kind
- No register allocation
- We have implemented a new compiler infrastructure which performs register allocation
- Still a one pass compiler
- Forms the basis for further native code optimizations

Object Heap Scalability

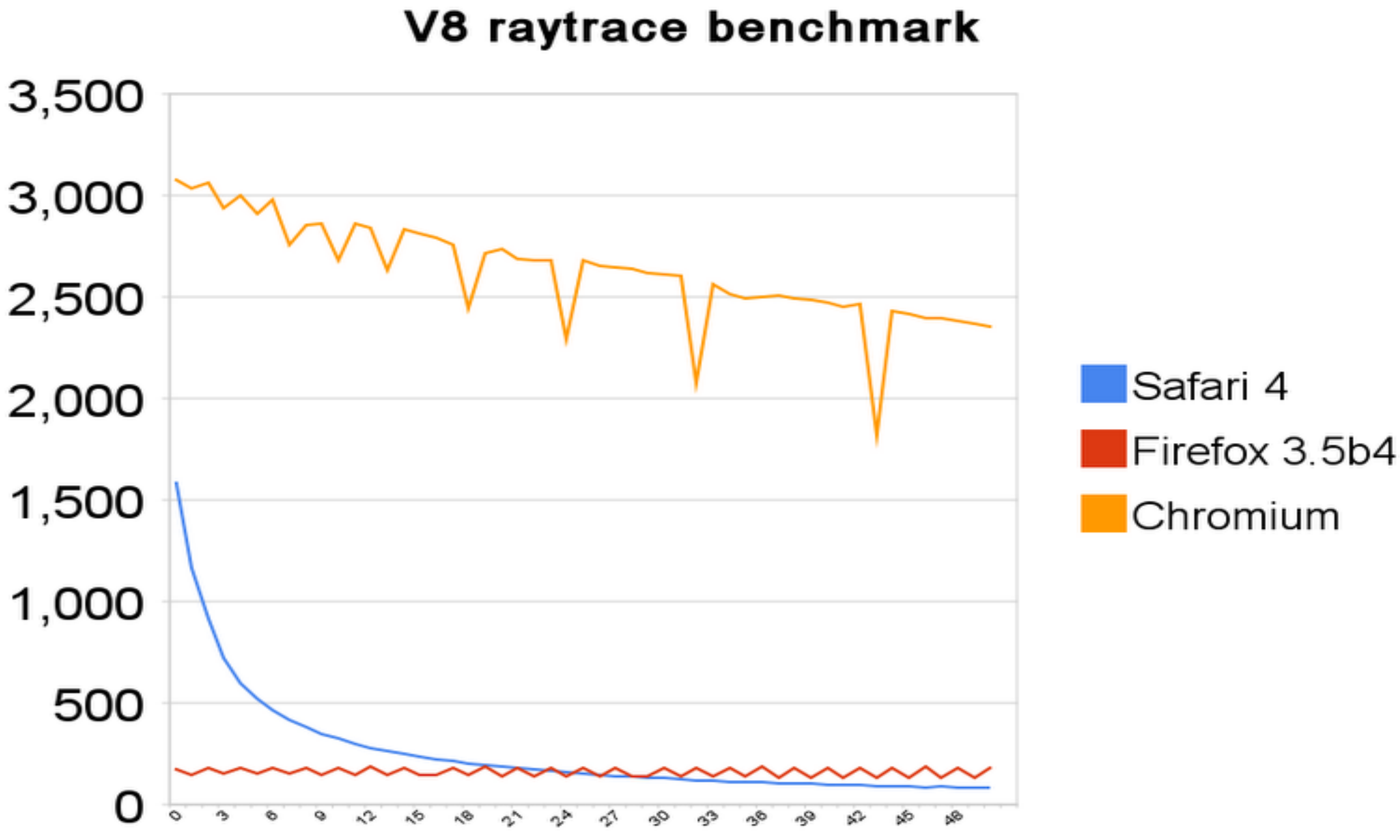
Scalability

- Users use multiple tabs running full applications
    - Mail
    - Calendar
    - News
- Applications are becoming bigger with more objects
- JavaScript execution should be fast in these situations
- The challenge is to scale well with respect to the size of the object heap
- The key to scaling well is generational garbage collection

Scalability Experiment

- Artificial scalability experiment approximating this situation
  - Raytrace benchmark from the V8 benchmark suite
  - Allocate extra live data on the side
  - 1MB of extra data per iteration

Scalability Experiment - Execution Speed



Bigger is better!

Scalability

- This experiment is artificial

- Try loading GMail in different browsers and then run JavaScript benchmarks in another tab

- Try out your own scalability experiments!

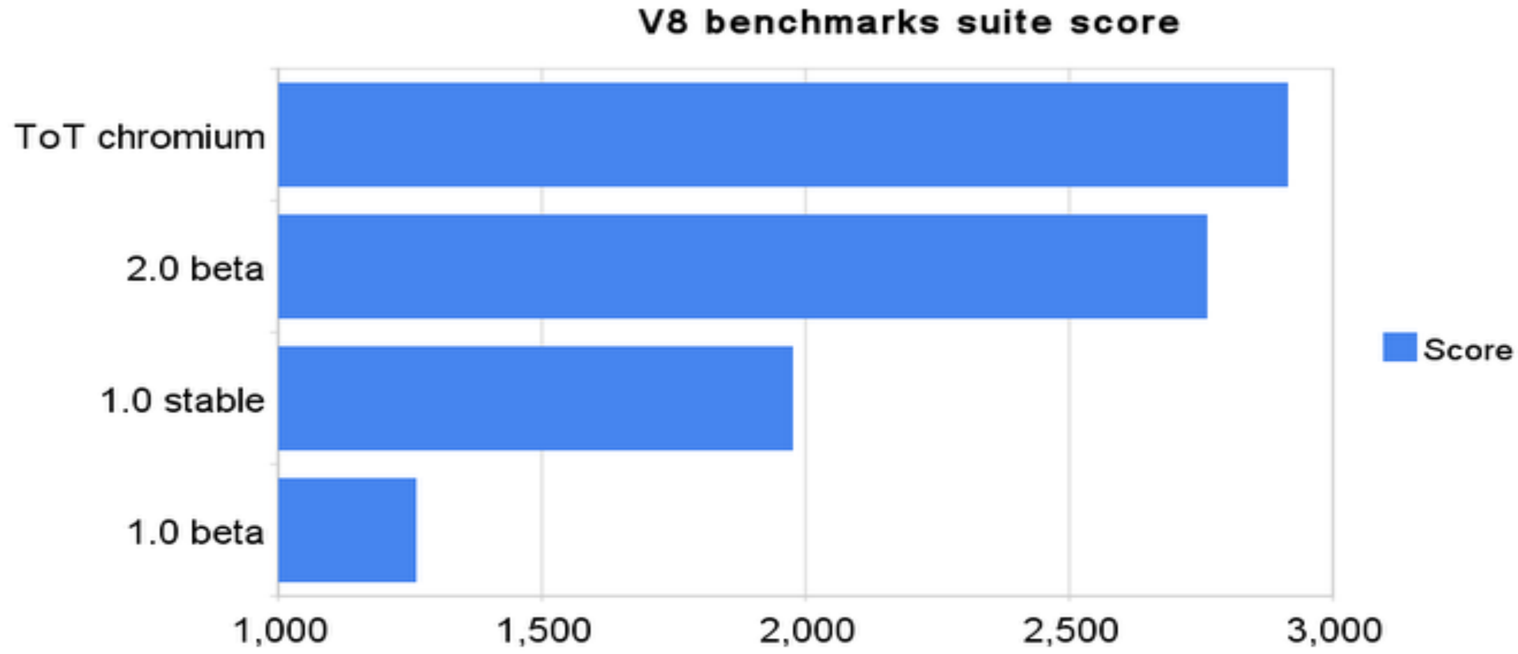
Performance Bottlenecks

Performance bottlenecks

- One fully general version of generated code
  - Generate optimized versions with more assumptions that falls back to fully general code
- Inline caching based on calls to stubs
  - Inline the fast common case directly and avoid the calls
- Slow write barrier
  - Experiment with other implementations
- No special handling of the global object
  - Loading of global properties can be much faster if we generate context specific code for global loads

Summary

- V8 was designed for speed and scalability
- The goal is to raise the performance bar for JavaScript



Google™ 09 IO