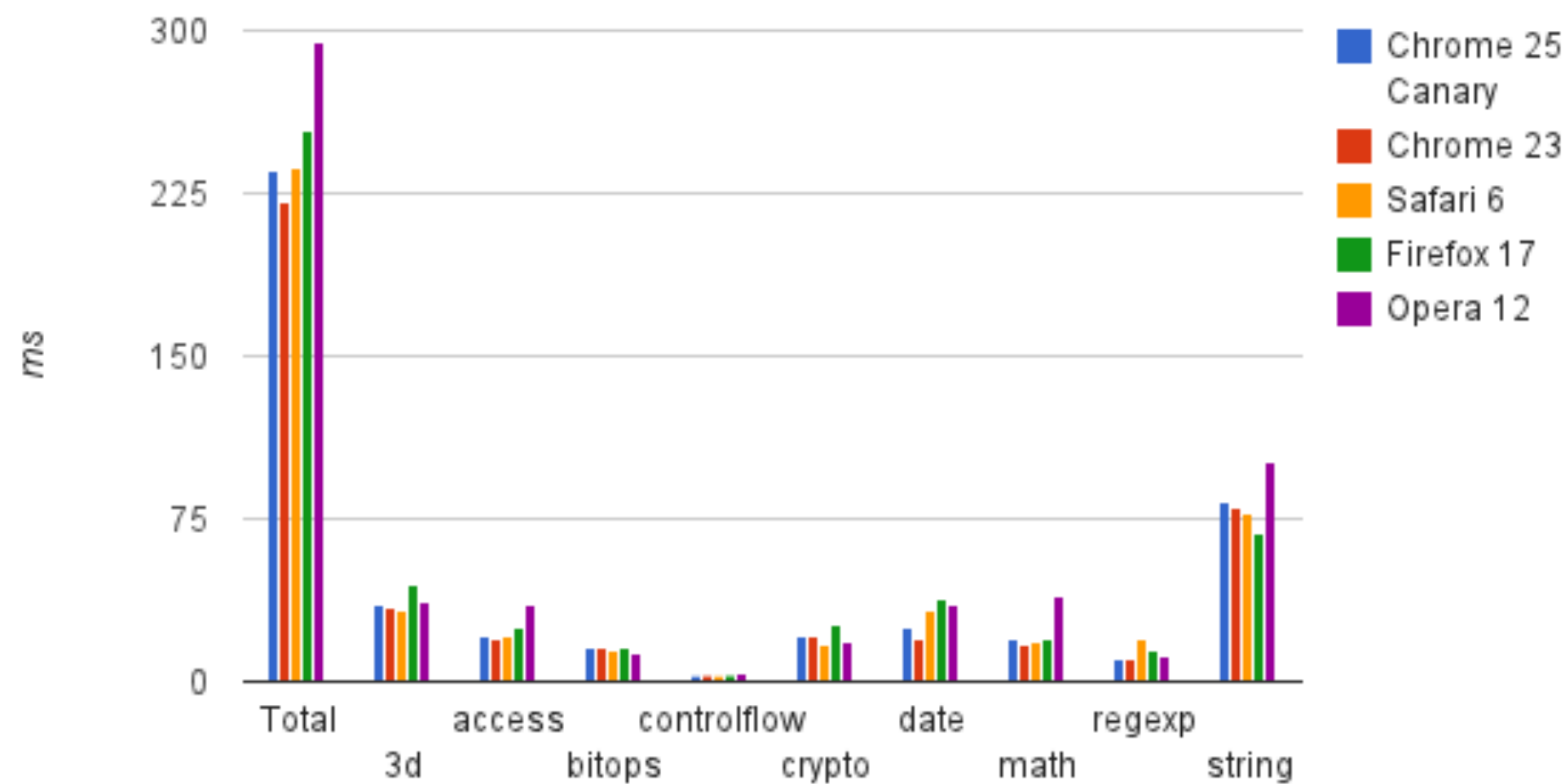# JavaScript Profiling and Optimization on V8
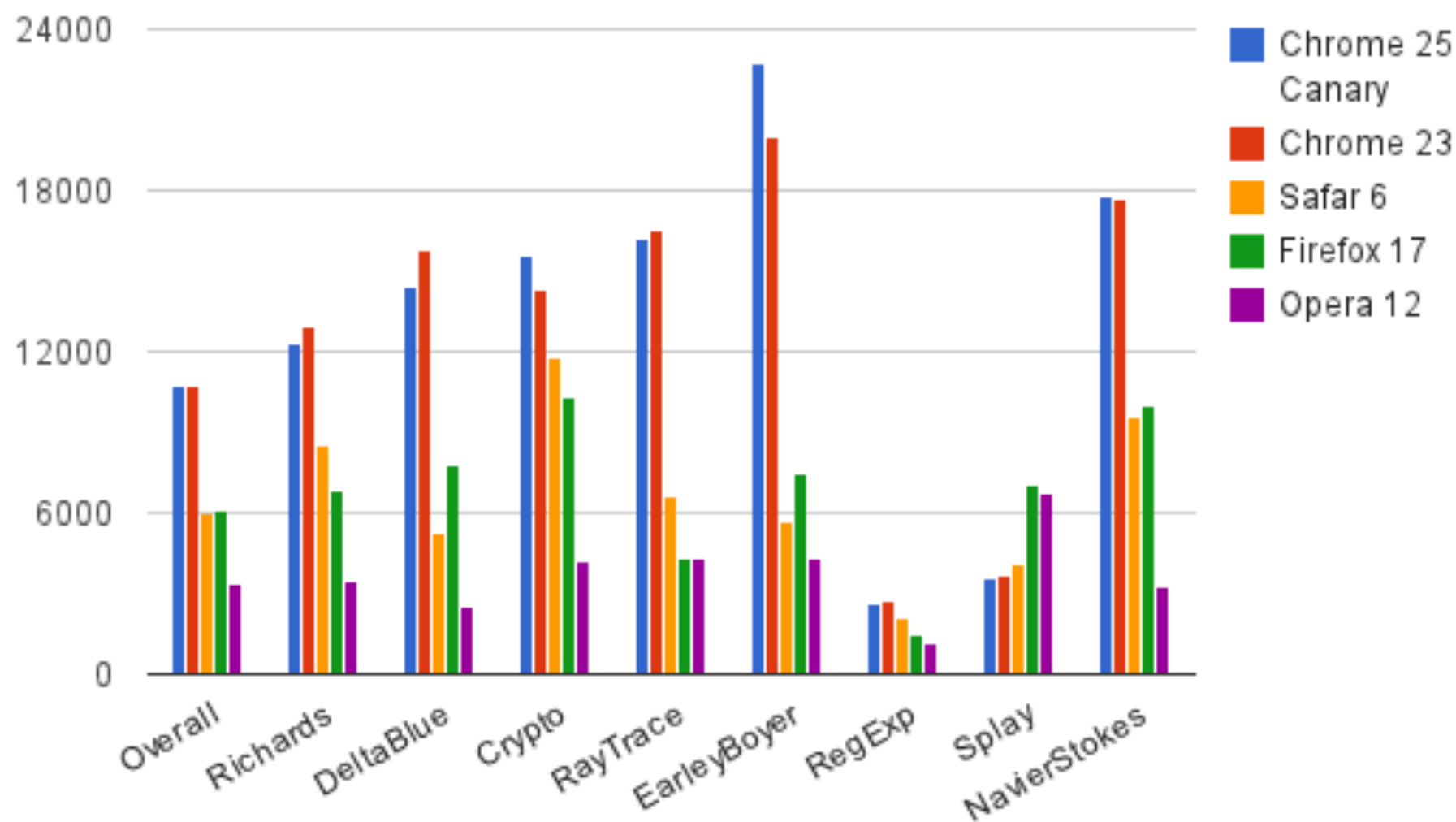
Yilin Zhang   C. Vic Hu

About JavaScript

- ECMAScript

- Used in Adobe Flash, Mac OS Dashboard widgets, Yahoo! widgets, browser bookmarklets, etc.

- Client-side scripting, DOM manipulation, Ajax

SunSpider 0.9.1 (smaller is better)

Legend:
- Chrome 25 Canary (blue)
- Chrome 23 (red)
- Safari 6 (orange)
- Firefox 17 (green)
- Opera 12 (purple)

Y-axis (ms): 0, 75, 150, 225, 300

X-axis categories: Total, 3d, access, bitops, controlflow, crypto, date, math, regexp, string

V8 Benchmark Suite v7 (bigger is better)

V8

- Open source JS engine by Google

- Written in C++ and used in Chrome

- Implements ECMAScript (ECMA-262, 5E)

- Can run standalone or embedded in any C++ applications

- Designed to be fast and efficient, like a V8 engine should be

What makes V8 so fast?

- Fast property access

- Dynamic machine code

- Efficient garbage collection

Fast Property Access

- JS is a dynamic language

- Hidden class vs. dictionary lookup

- High degree of structure-sharing

Fast Property Access

- JS is a dynamic language

- **Hidden class** vs. dictionary lookup

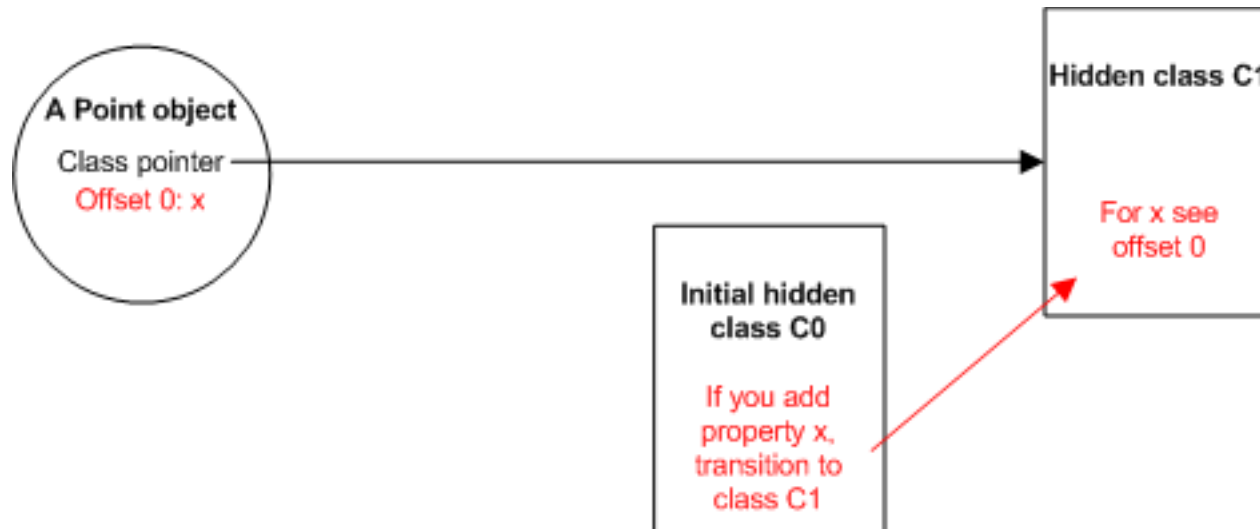- High degree of structure-sharing

Hidden Class

- Created dynamically behind the scene

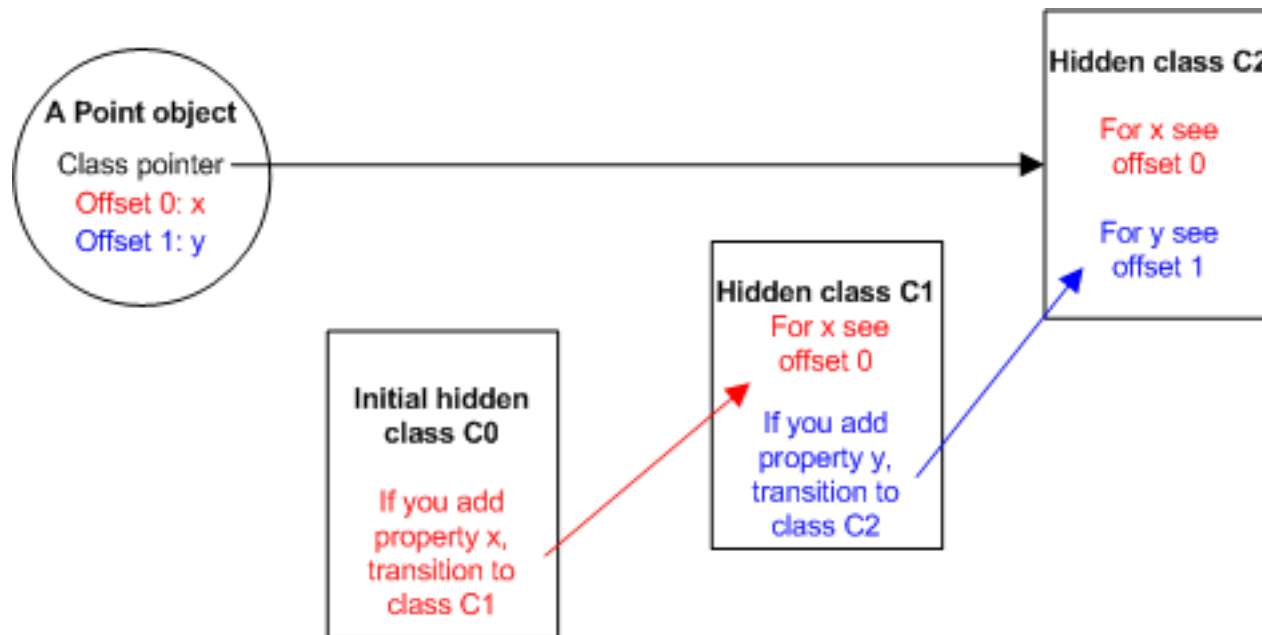- Changed when a new property is added

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

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

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

A Point object

Class pointer ———————————————————————→ **Hidden class C1**

Offset 0: x

Initial hidden class C0

For x see offset 0

If you add property x, transition to class C1

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


A Point object
Class pointer
Offset 0: x
Offset 1: y

Hidden class C2
For x see offset 0
For y see offset 1

Hidden class C1
For x see offset 0
If you add property y, transition to class C2

Initial hidden class C0
If you add property x, transition to class C1

Advantages of using hidden class

- No dictionary lookup for property access

- Enable V8 to use inline caching

Inline Cache (IC)

- Caches type-dependent code

- Needs to validate type assumptions

- Changed at runtime

Dynamic Machine Code

- Directly compiled into machine code

- No intermediate byte code or interpreter

- Inline caching for correct prediction on hidden classes

Efficient Garbage Collection

- Stops program execution

- Processes only part of the object heap

- Accurate objects and pointers locating in memory to avoid memory leaks

V8 Compilation Process

1. Base compiler

2. Runtime profiler

3. Optimizing compiler

4. Deoptimization support

1. Base Compiler

- A full compiler to generate machine code upon execution

- No assumption about types at compilation time

- Uses ICs to retrieve types at runtime

- Fast, but not optimized

2. Runtime Profiler

- Sampling every milliseconds

- In preparation for the optimization

3. Optimizing Compiler

- SSA form

- Loop-invariant code motion

- Common subexpression elimination

- Linear-scan register allocation

- Function inlining

4. Deoptimization Support

- What if overly optimistic?

- An opportunity to 'bail-out'

- Easily observed with the profiler

JavaScript is Slow

Problem:

Compute the 25,000th prime

Algorithm:

For x = 1 to infinity: if x not divisible
  by any member of an initially empty list
  of primes, add x to the list until we
  have 25,000

The Contenders

c++

```cpp
class Primes {
    public:
      int getPrimeCount() const { return prime_count; }
      int getPrime(int i) const { return primes[i]; }
      void addPrime(int i) { primes[prime_count++] = i; }

      bool isDivisibe(int i, int by) { return (i % by) == 0; }

      bool isPrimeDivisible(int candidate) {
        for (int i = 1; i < prime_count; ++i) {
          if (isDivisibe(candidate, primes[i])) return true;
        }
        return false;
      }

    private:
      volatile int prime_count;
      volatile int primes[25000];
};int main() {
  Primes p;
  int c = 1;
  while (p.getPrimeCount() < 25000) {
    if (!p.isPrimeDivisible(c)) {
      p.addPrime(c);
    }
    c++;
  }
  printf("%d\n", p.getPrime(p.getPrimeCount()-1));
}
```

JavaScript

```javascript
function Primes() {
  this.prime_count = 0;
  this.primes = new Array(25000);
  this.getPrimeCount = function() { return this.prime_count; }
  this.getPrime = function(i) { return this.primes[i]; }
  this.addPrime = function(i) {
    this.primes[this.prime_count++] = i;
  }

  this.isPrimeDivisible = function(candidate) {
    for (var i = 1; i <= this.prime_count; ++i) {
      if ((candidate % this.primes[i]) == 0) return true;
    }
    return false;
  }
};function main() {
  p = new Primes();
  var c = 1;
  while (p.getPrimeCount() < 25000) {
    if (!p.isPrimeDivisible(c)) {
      p.addPrime(c);
    }
    c++;
  }
  print(p.getPrime(p.getPrimeCount()-1));
}

main();
```

```cpp
class Primes {
  public:
    int getPrimeCount() const { return prime_count; }
    int getPrime(int i) const { return primes[i]; }
    void addPrime(int i) { primes[prime_count++] = i; }

    bool isDivisibe(int i, int by) { return (i % by) == 0; }

    bool isPrimeDivisible(int candidate) {
      for (int i = 1; i < prime_count; ++i) {
        if (isDivisibe(candidate, primes[i])) return true;
      }
      return false;
    }
  private:
    volatile int prime_count;
    volatile int primes[25000];
};int main() {
    Primes p;
    int c = 1;
    while (p.getPrimeCount() < 25000) {
      if (!p.isPrimeDivisible(c)) {
        p.addPrime(c);
      }
      c++;
    }
    printf("%d\n", p.getPrime(p.getPrimeCount()-1));
}
```

```javascript
function Primes() {
    this.prime_count = 0;
    this.primes = new Array(25000);
    this.getPrimeCount = function() { return this.prime_count; }
    this.getPrime = function(i) { return this.primes[i]; }
    this.addPrime = function(i) {
      this.primes[this.prime_count++] = i;
    }

    this.isPrimeDivisible = function(candidate) {
      for (var i = 1; i <= this.prime_count; ++i) {
        if ((candidate % this.primes[i]) == 0) return true;
      }
      return false;
    }
};function main() {
    p = new Primes();
    var c = 1;
    while (p.getPrimeCount() < 25000) {
      if (!p.isPrimeDivisible(c)) {
        p.addPrime(c);
      }
      c++;
    }
    print(p.getPrime(p.getPrimeCount()-1));
}main();
```

The Results

```
% g++ primes.cc -o primes
  % time ./primes287107


  real   0m2.955s
  user   0m2.952s        c++
  sys    0m0.001s
```

```
% time d8 primes.js287107

real       0m15.584s
user       0m15.612s
sys        0m0.073s         JavaScript
```

C++ is about 5x faster than JavaScript

Inline Caches

Candidate % this.primes[i]

No calls in this code

Inline Caches

Candidate % this.primes[i]

this.primes[i]

->call LoadIC_Initialize

-> call 0x311286e0

-> move eax, [edi+0fx]

Logging What Gets Optimized

Command:

./out/ia32.release/d8 --trace-opt samples/
  primes.js

Log name of optimized functions to stdout:

addPrime, IsPrimDivisible, main...

Profiling the JavaScripts

```
./out/ia32.release/d8 samples/primes.js --
    prof
  287107

./tools/mac-tick-processor v8.log
```

Profiling the JavaScripts

Statistical profiling result from v8.log, (10868 ticks, 81 unaccounted, 0 excluded).
[JavaScript]:
```
 ticks   total   nonlib    name
 1254   11.5%    11.5%    LazyCompile: *main samples/primes.js:18
  959    8.8%     8.8%    LazyCompile: MOD native runtime.js:238
  643    5.9%     5.9%    Stub: CEntryStub
  468    4.3%     4.3%    KeyedLoadIC: A keyed load IC from the snapshot
  388    3.6%     3.6%    Stub: BinaryOpStub_MOD_Alloc_SMI+Oddball
    1    0.0%     0.0%    LazyCompile: ~Primes.isPrimeDivisible samples/primes.js:10
```
[C++]:
```
 ticks   total   nonlib    name
 3274   30.1%    30.1%    _atanhl$fenv_access_off
 1301   12.0%    12.0%    v8::internal::Runtime_NumberMod
  979    9.0%     9.0%    v8::internal::Heap::NumberFromDouble
```

Statistical profiling result from v8.log, (10868 ticks, 81 unaccounted, 0 excluded).

[JavaScript]:

ticks	total	nonlib	name
1254	11.5%	11.5%	LazyCompile: *main samples/primes.js:18
959	8.8%	8.8%	LazyCompile: MOD native runtime.js:238
643	5.9%	5.9%	Stub: CEntryStub
468	4.3%	4.3%	KeyedLoadIC: A keyed load IC from the snapshot
388	3.6%	3.6%	Stub: BinaryOpStub_MOD_Alloc_SMI+Oddball
1	0.0%	0.0%	LazyCompile: ~Primes.isPrimeDivisible samples/primes.js:10

[C++]:

ticks	total	nonlib	name
3274	30.1%	30.1%	_atanl$fenv_access_off
1301	12.0%	12.0%	v8::internal::Runtime_NumberMod
979	9.0%	9.0%	v8::internal::Heap::NumberFromDouble

Something is Wrong with the Code

```
this.isPrimeDivisible = function(candidate) {
    for (var i = 1; i <= this.prime_count; ++i) {
      if ((candidate % this.primes[i]) == 0) return true;
    }
    return false;
}
```

Profiling Again

```
/out/ia32.release/d8 samples/primes.js --prof
```

287107

```
./tools/mac-tick-processor v8.log
```

Profiling Again

[JavaScript]:

```
ticks   total   nonlib    name
1426    99.4%   99.4%   LazyCompile: *main samples/primes-2.js:18
   5     0.3%    0.3%   LazyCompile: *Primes.isPrimeDivisible samples/primes-2.js:
 10
```

[C++]:

```
ticks   total   nonlib    name
   1     0.1%    0.1%   v8::internal::StaticVisitorBase::GetVisitorId
   1     0.1%    0.1%   v8::internal::Runtime_FunctionSetName
   1     0.1%    0.1%   v8::internal::Map::LookupDescriptor
   1     0.1%    0.1%   v8::internal::LAllocator::TraceAlloc
```

```
[JavaScript]:
  ticks   total   nonlib    name
  1426   99.4%   99.4%   LazyCompile: *main samples/
  primes-2.js:18

     5    0.3%    0.3%   LazyCompile:
  *Primes.isPrimeDivisible samples/primes-2.js:10


[C++]:
  ticks   total   nonlib    name
     1    0.1%    0.1%
  v8::internal::StaticVisitorBase::GetVisitorId
     1    0.1%    0.1%
  v8::internal::Runtime_FunctionSetName
     1    0.1%    0.1%
  v8::internal::Map::LookupDescriptor
     1    0.1%    0.1%
  v8::internal::LAllocator::TraceAlloc
```

Runtime

```
time v8 primes-2.js
287107
real 0m1.829s
user 0m1.827s
sys  0m0.010s
```

Optimize Your Algorithm

```javascript
this.isPrimeDivisible = function(candidate) {
    for (var i = 1; i < this.prime_count; ++i) {
  var current_prime = this.primes[i];
      if (current_prime * current_prime > candidate) {
        return false;
      }
  if ((candidate % current_prime) == 0) return true;
  }
  return false;
}
```

Optimize Your Algorithm

```
time v8 primes-3.js
287107
real 0m0.044s
user 0m0.038s
sys  0m0.004s
```

Crankshaft

Hydrogen-SSA in Crankshaft [Wingolog]

1. To permit inlining. Inlining is truly the mother of all optimizations, in that it permits more optimizations to occur.

2. To facilitate loop-invariant code motion and common subexpression elimination.

SSA Data Structures

HGraph

HBasicBlock* blocks[];
HPhi* all_phis[]; // facilitates analysis
HValue* all_values[]; // allows lookup by index

HInstruction : HValue

HInstruction *next, *prev;

HBasicBlock

int id;
HGraph* graph;
HPhi* phis[]
HInstruction* first;
HControlInstruction* last;
HLoopInformation* loop_info;
HBasicBlock* predecessors[];
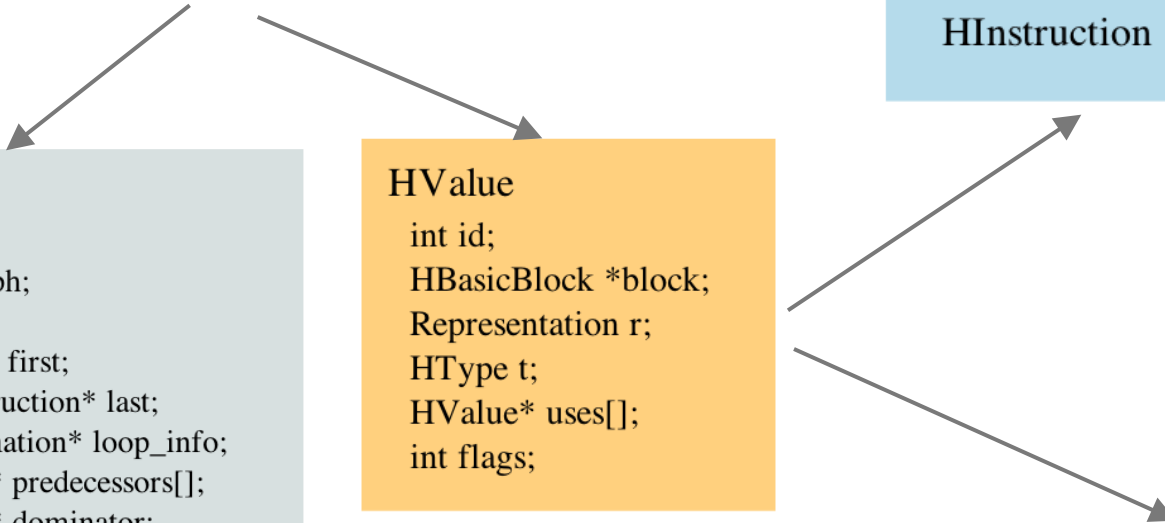HBasicBlock* dominator;
HBasicBlock* dominated[];

HValue

int id;
HBasicBlock *block;
Representation r;
HType t;
HValue* uses[];
int flags;

HPhi : HValue

HValue* inputs[];

Optimization Proposal

- More sophisticated algorithms in determining the 'hot code': frequency alone might not be the best way to define 'hot'
    - Use counter instead of sampling every millisecond

- Phi node insertion optimization

Conclusion

- Chrome with V8 engine outperforms other browsers in most benchmarks, especially V8 benchmarks

- Hidden classes introduces inline caching and type assumption

- Use run-time profiling to optimize hot code, and do nothing if the profiler decided it's not hot