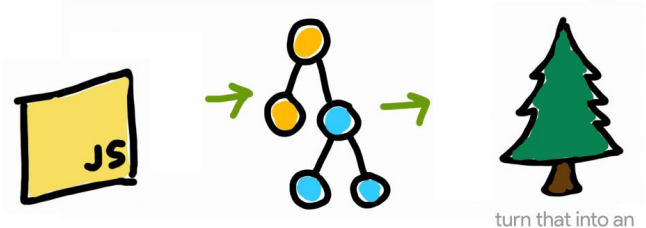
Google

TurboFan: A new code generation architecture for V8

Benedikt Meurer
Google Munich
@bmeurer

How

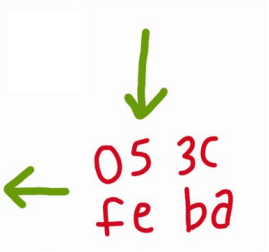
Works



We take your JS

Parse it

turn that into an Abstract Syntax Tree

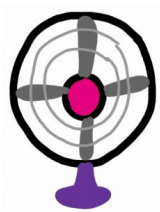


Generate bytecode

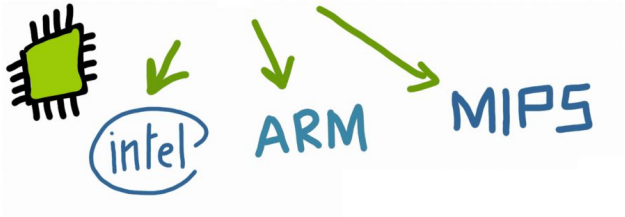
Get feedback

for speculative optimisations

TURBOFAN



Optimize & Compile it



then run your optimized code!

By @addyosmani

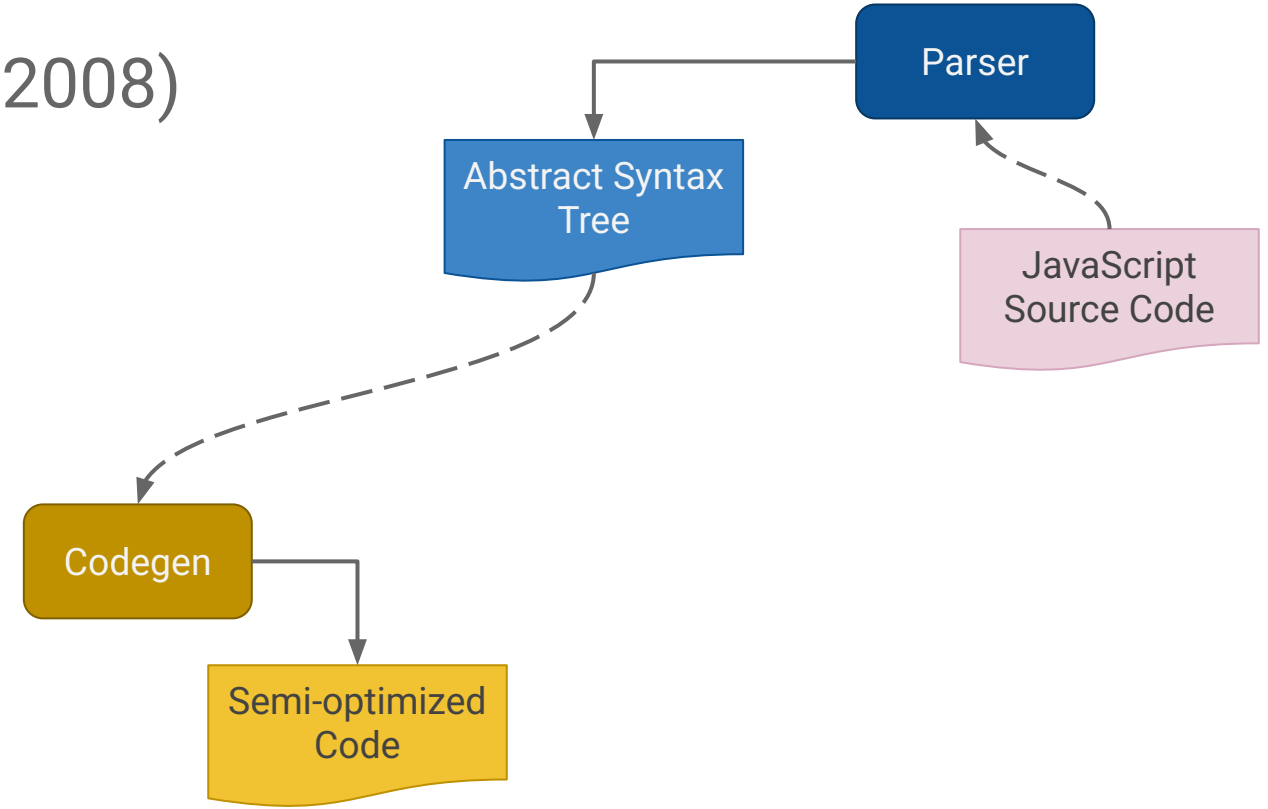
Source: medium.com/dev-channel/javascript-start-up-performance-69200f43b201

Why a new code generation architecture?

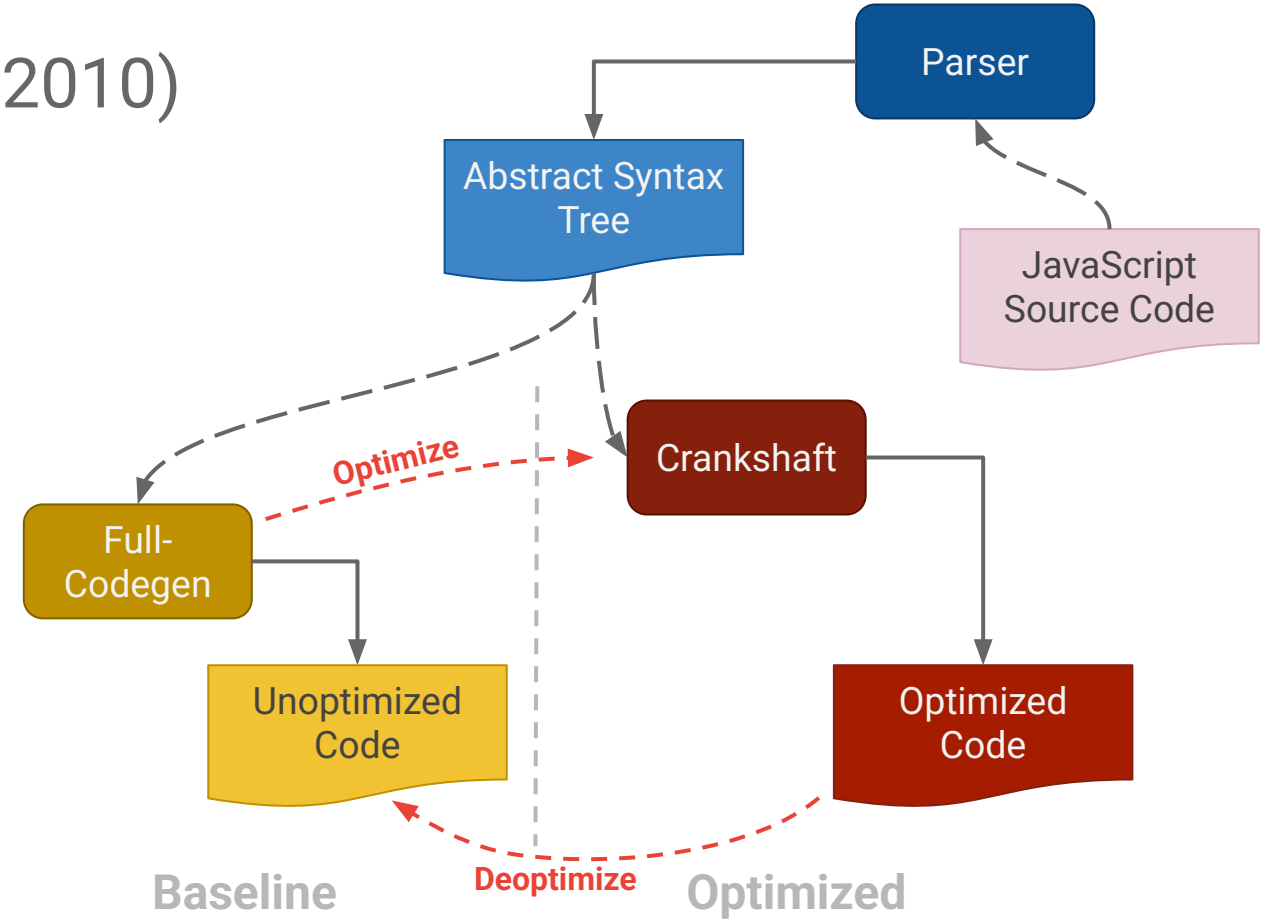
- Improve baseline performance
- Make performance predictable
- Reduce page load time
- Reduce memory usage
- Reduce complexity


A bit of history...

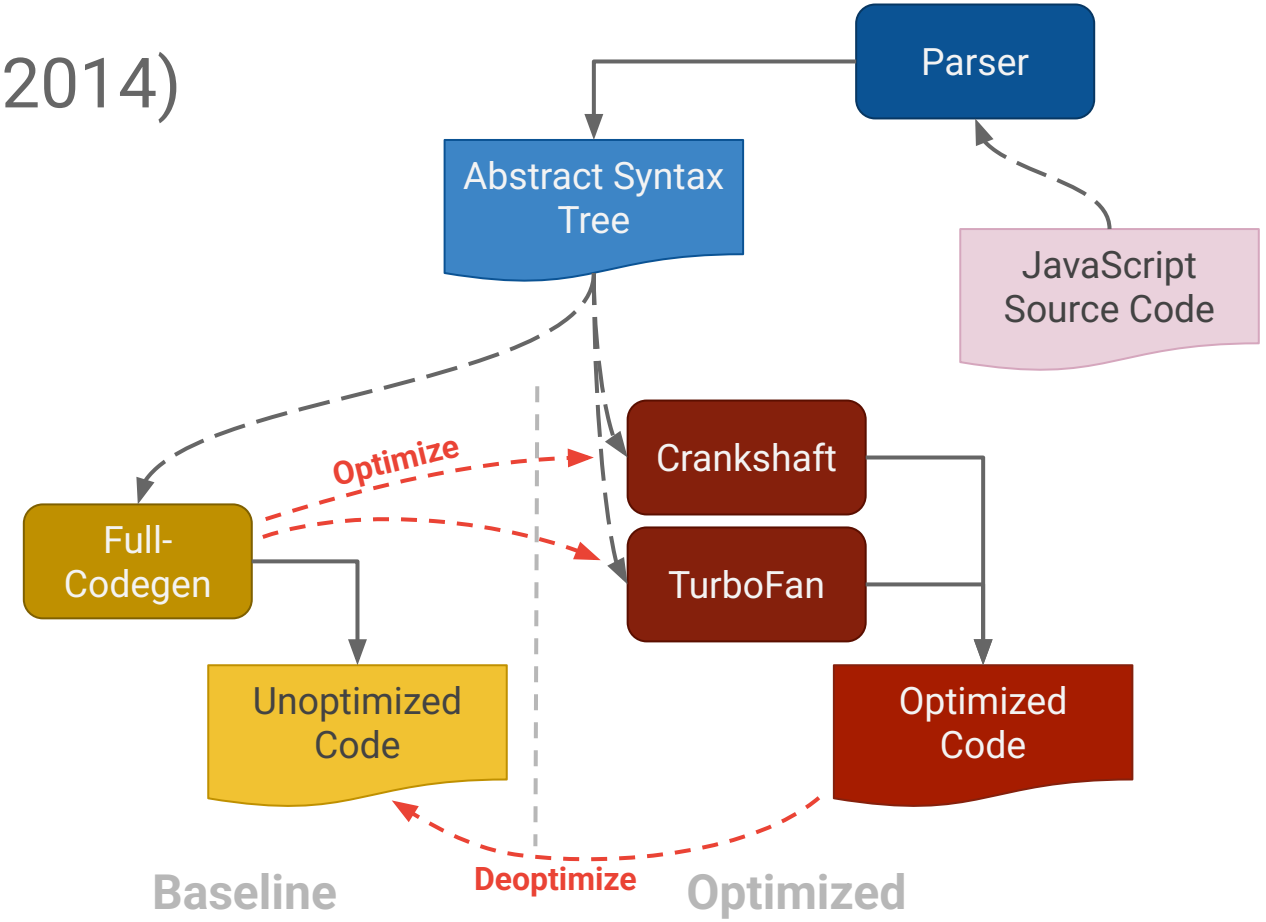
Compiler pipeline (2008)



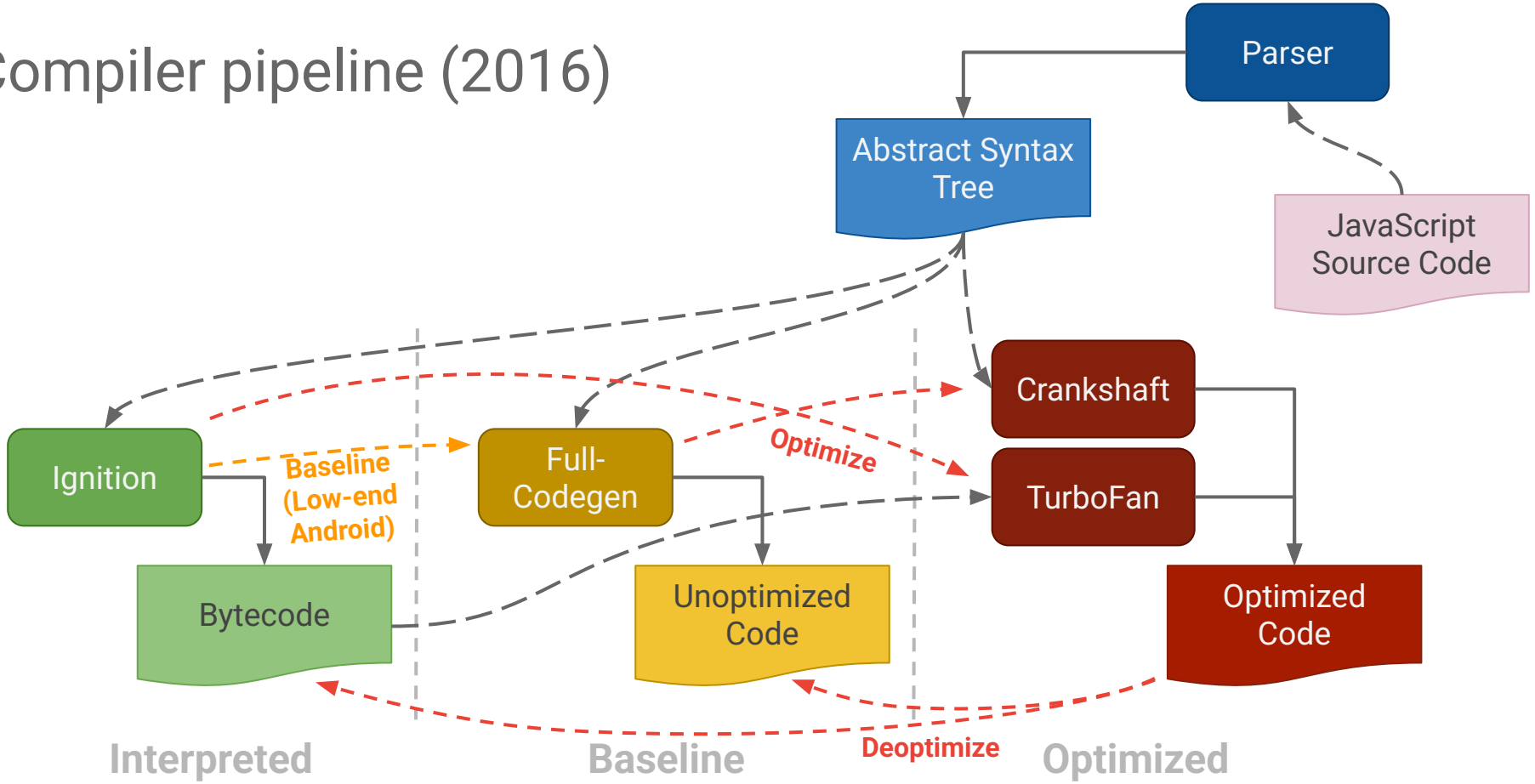
Compiler pipeline (2010)



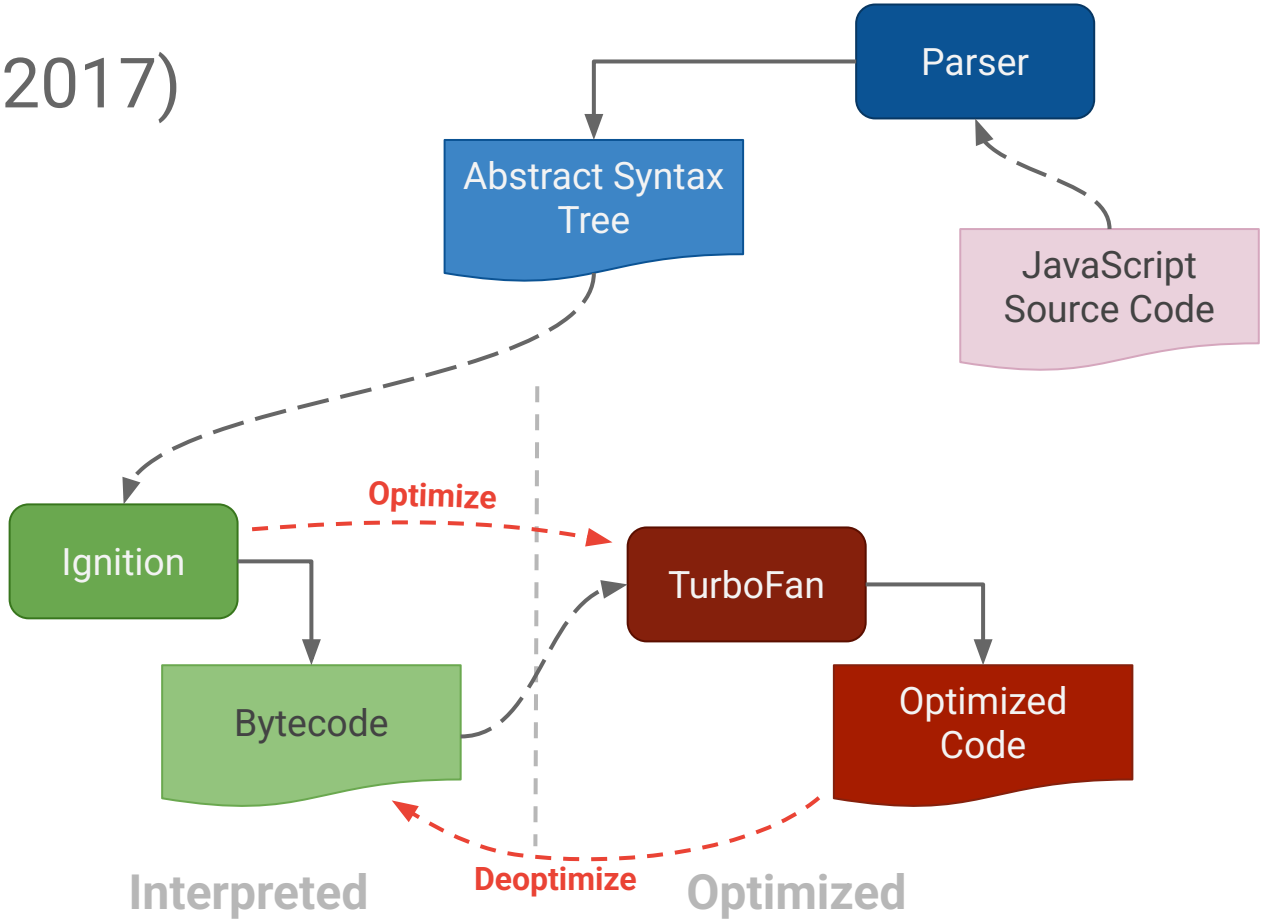
Compiler pipeline (2014)



Compiler pipeline (2016)

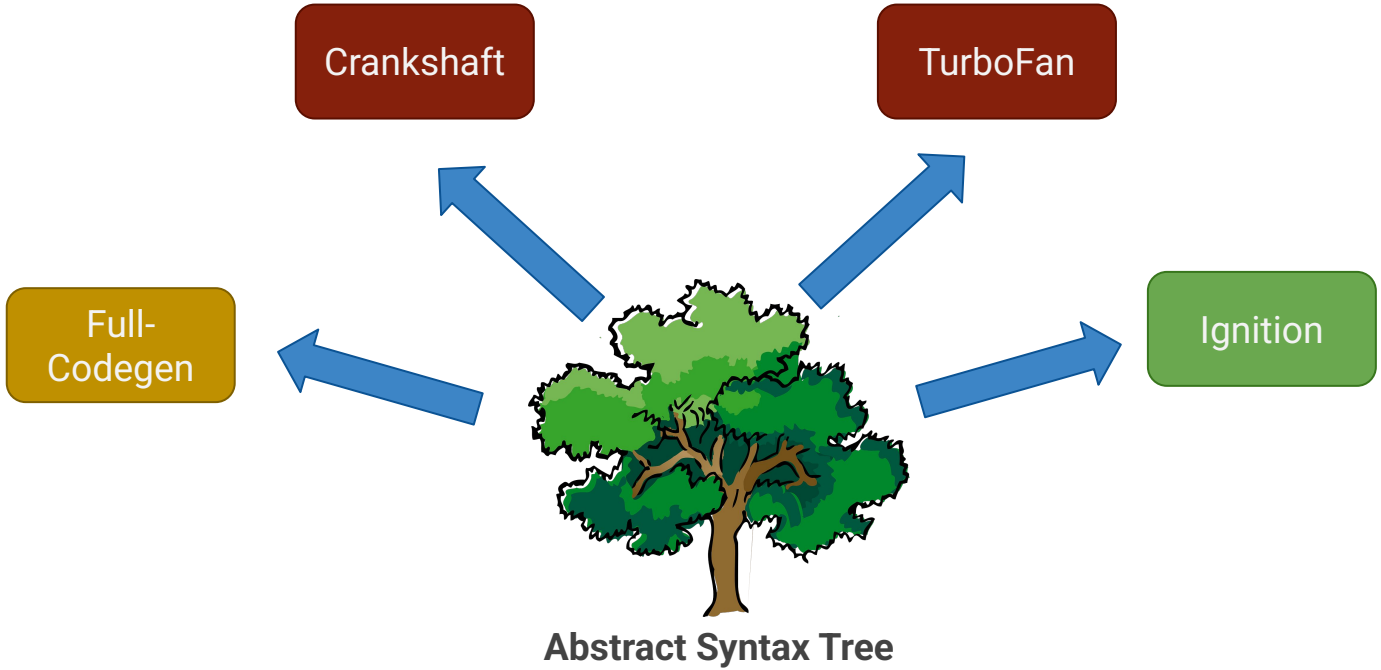


Compiler pipeline (2017)



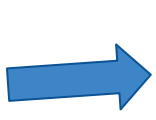
Reduce complexity

Abstract Syntax Tree

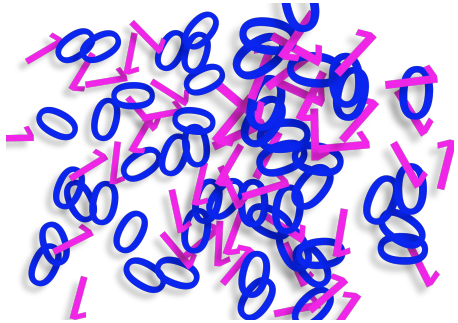


Abstract Syntax Tree

Abstract Syntax Tree

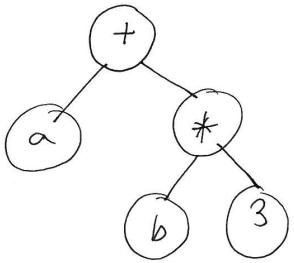


Ignition



Bytecode

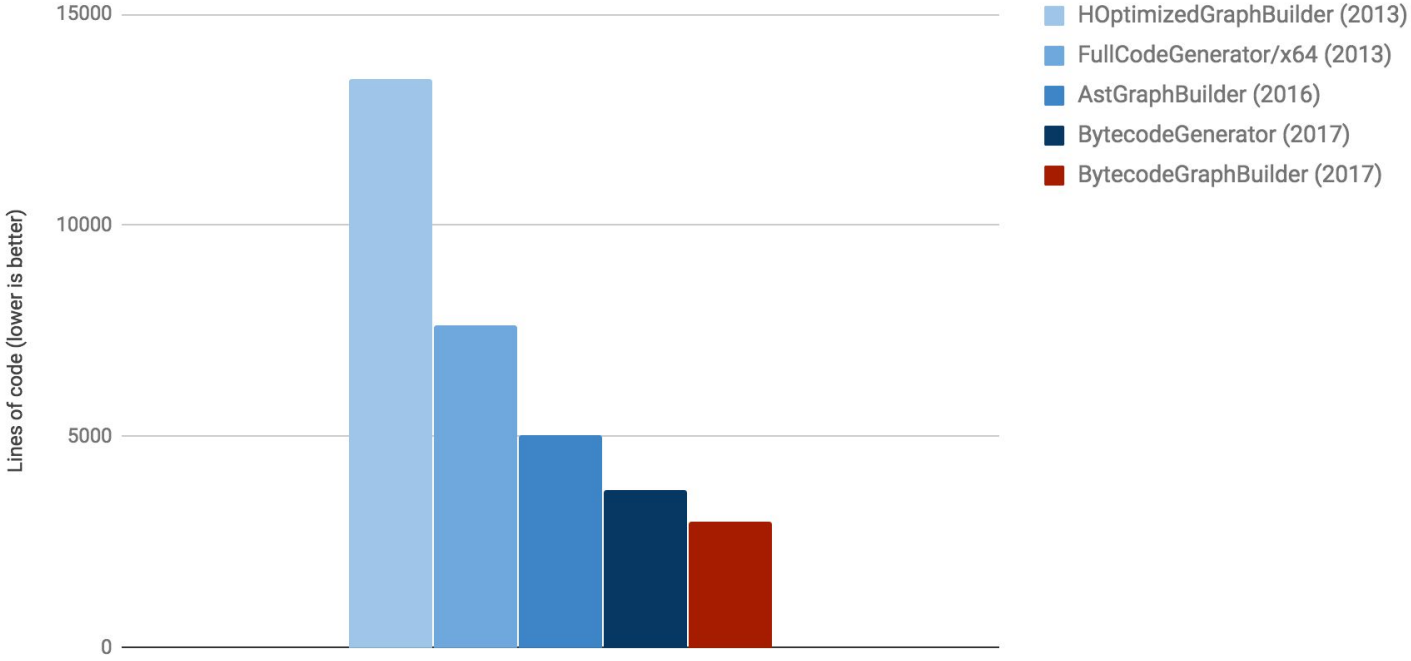
TurboFan



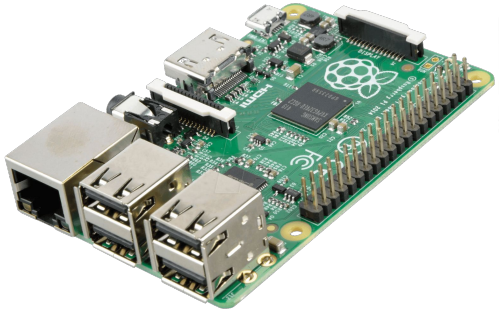

```
0 : 87        StackCheck
1 : 03 03     LdaSmi [3]
3 : 2e 02 02  Mul a1, [2]
6 : 2c 03 03  Add a0, [3]
9 : 8b        Return
```

```
const f = (a, b) => a + b * 3;
```

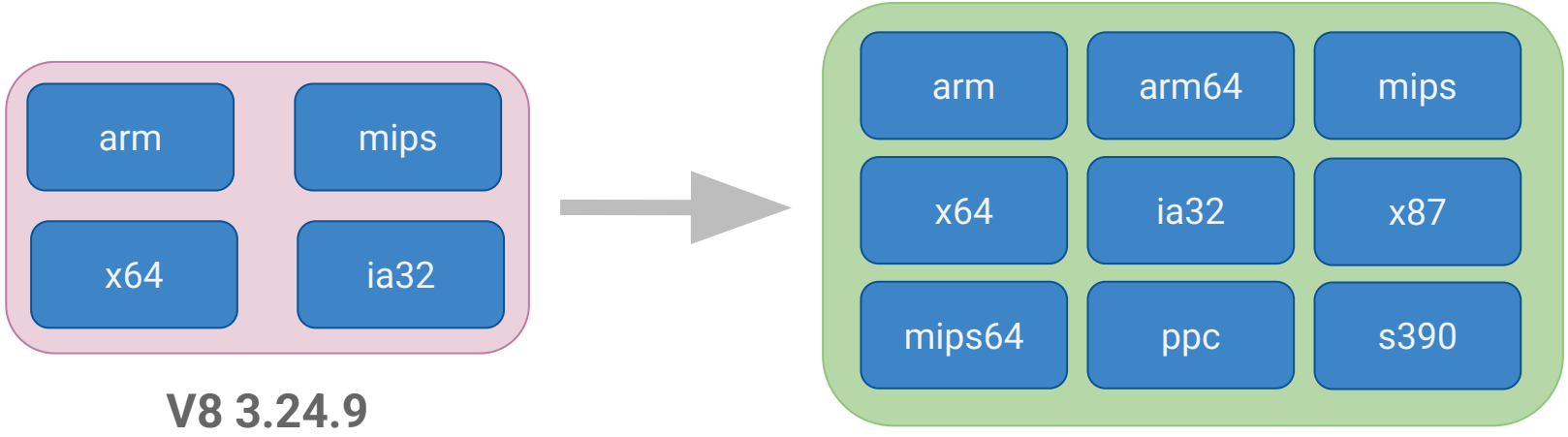
Abstract Syntax Tree



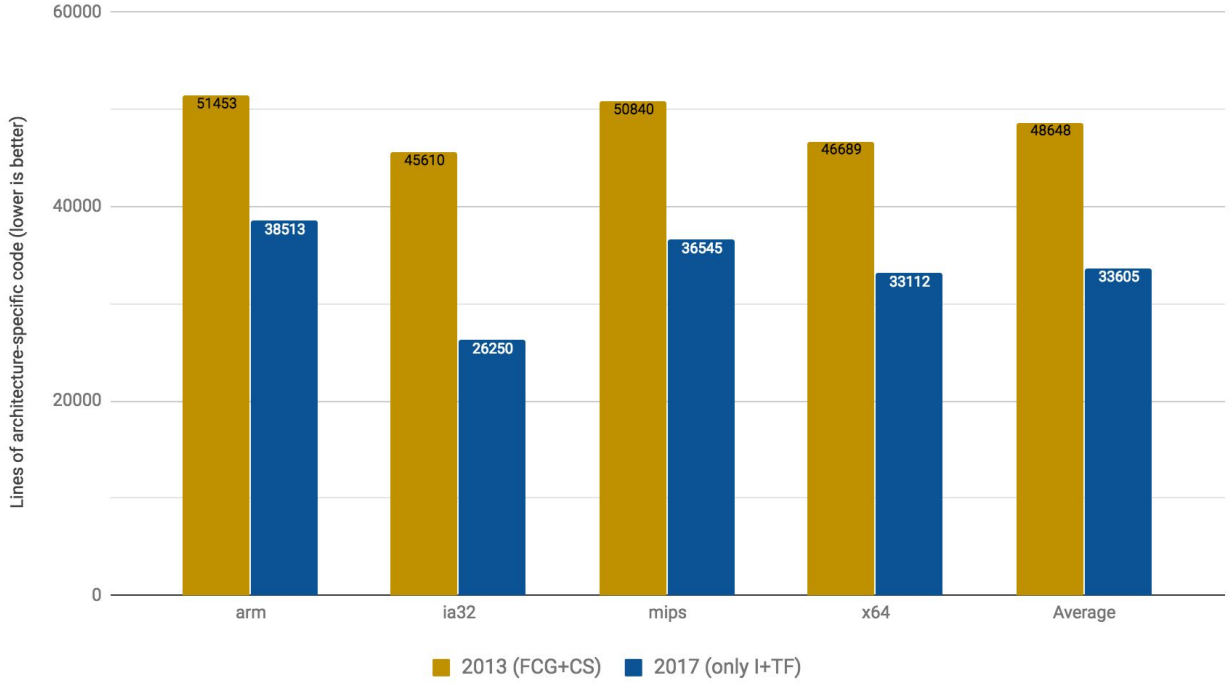
Supported architectures



Supported architectures



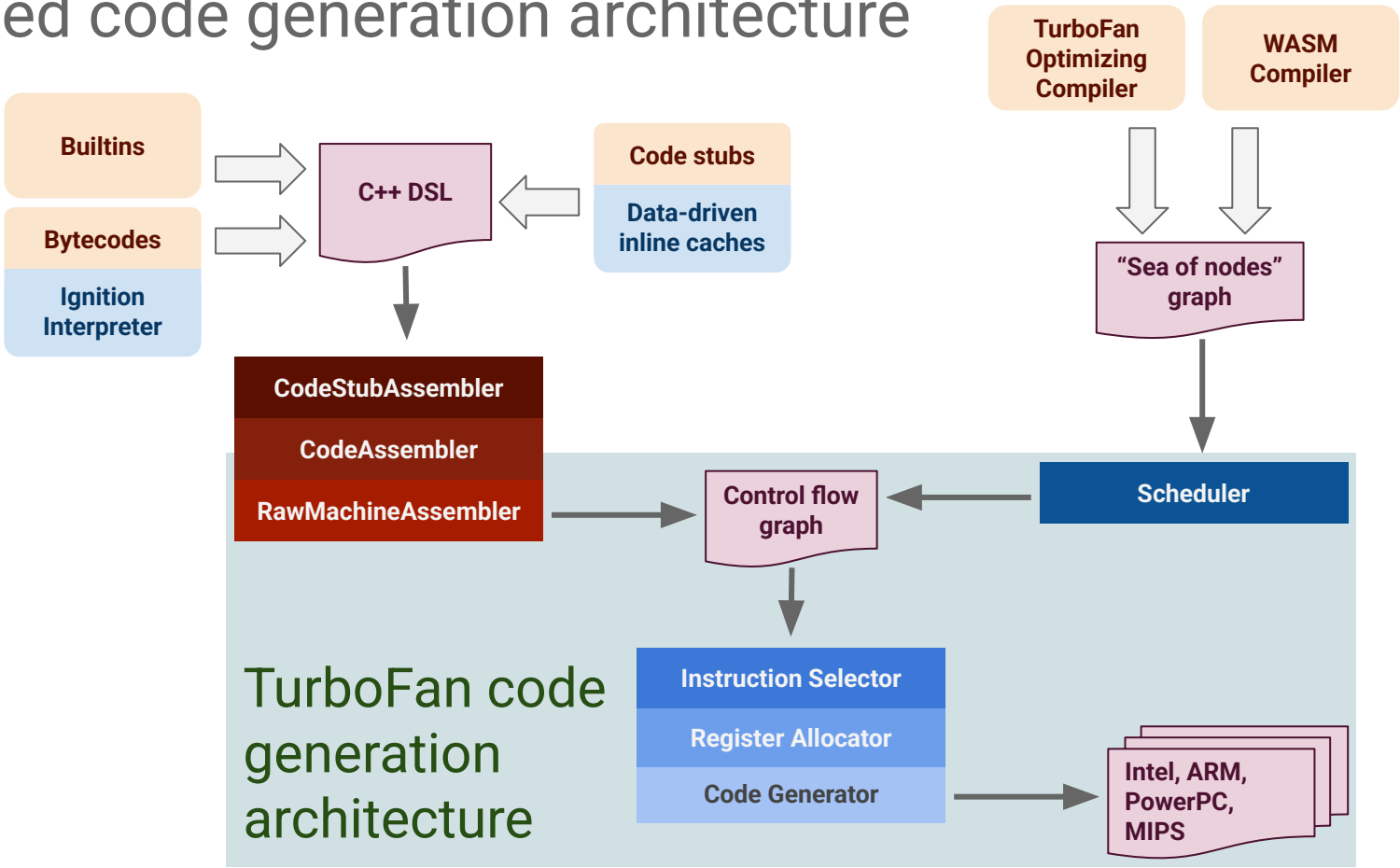
Architecture-specific LOCs



29%

less architecture-specific code from 2013 to 2017

Unified code generation architecture



Unified code generation architecture

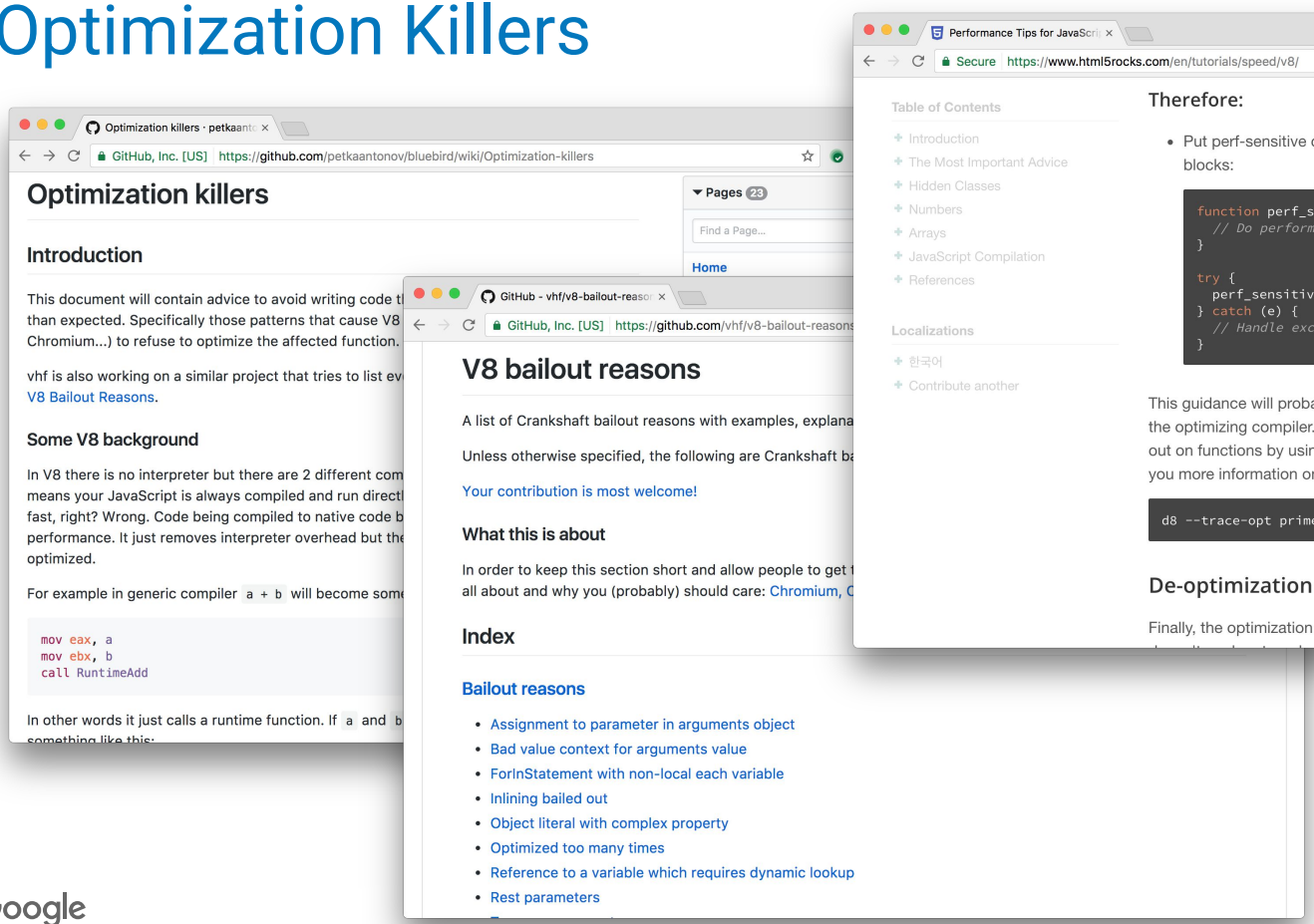
- (not just an) Optimizing compiler
- Interpreter bytecode handlers
- Builtins (Object.create, Array.prototype.indexOf, etc.)
- Code stubs / IC subsystem
- WebAssembly code generation

Unified code generation architecture

- More opportunities for performance improvements
- Fewer bugs (no more register allocation by hand)
- Easier to port to new architectures

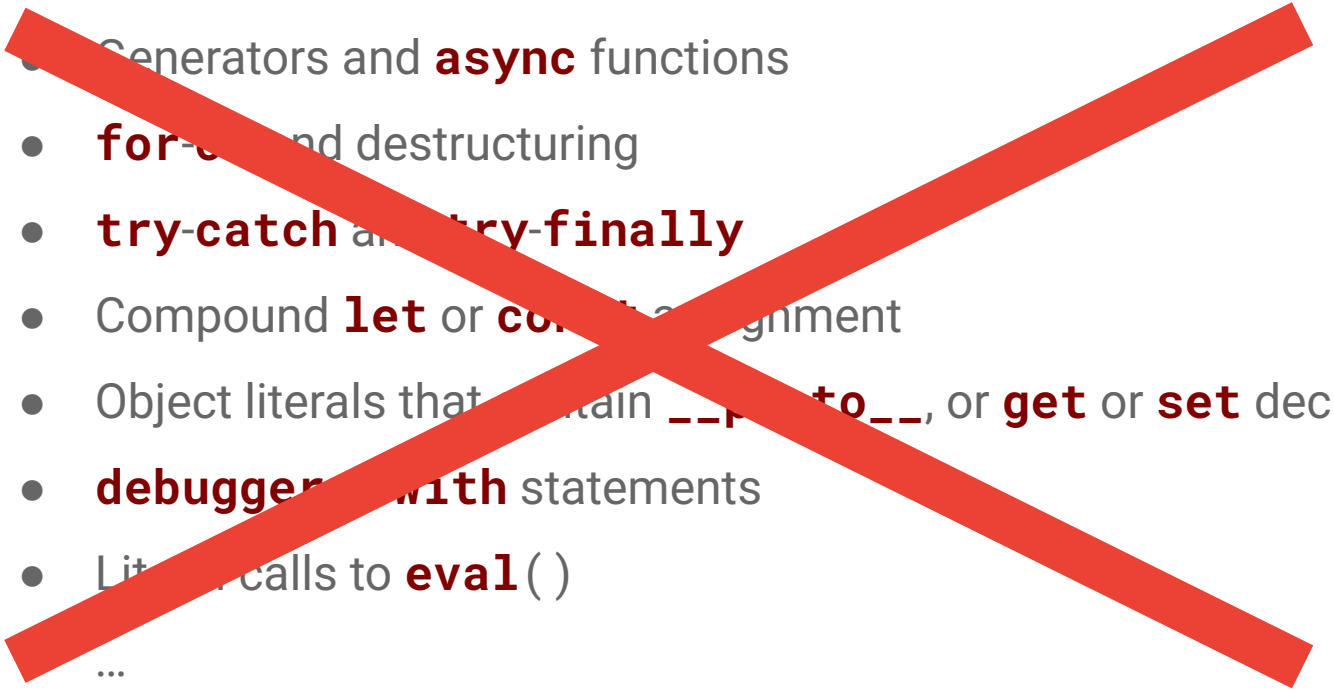
Predictable Performance

Optimization Killers



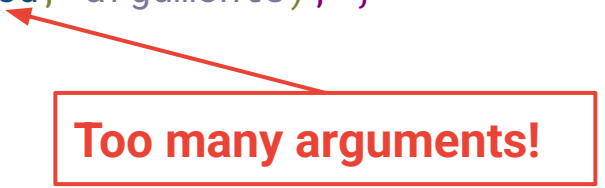
Optimization Killers

- Generators and **async** functions
- **for-** nd destructuring
- **try-catch** a **try-finally**
- Compound **let** or **co** gnment
- Object literals that tain **__p to__**, or **get** or **set** declarations.
- **debugger** **with** statements
- Li calls to **eval**( )
- …



Optimization Killers - arguments object

```
const arr = new Array(4000);
function mymax() { return Math.max.apply(undefined, arguments); }
for (let i = 0; i < 2000; ++i) mymax(...arr);
```

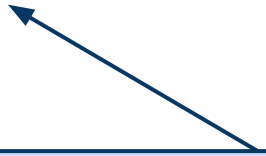



```
$ node --trace-opt --trace-deopt apply.js
…
[deoptimizing (DEOPT eager): begin 0x34e770317ea1 <JS Function mymax
(SharedFunctionInfo 0x2793cb1f4031)> (opt #5) @2, FP to SP delta: 24, caller sp:
0x7fff5fbf6378]
…
[disabled optimization for 0x2793cb1f4031 <SharedFunctionInfo mymax>, reason:
Optimized too many times]
$
```

Optimization Killers - arguments object

```
var callbacks = [
  function sloppy() {},
  function strict() { "use strict"; }
];

function dispatch() {
  for (var l = callbacks.length, i = 0; i < l; ++i) {
    callbacks[i].apply(null, arguments);
  }
}
```



[disabled optimization for … <SharedFunctionInfo `dispatch`>, reason: Bad value context for `arguments` value]

Optimization Killers

Setup

```
const N = 1000;
const arr = Array.apply(null, { length: N }).map(Number.call, Number);
let total = 0;
```

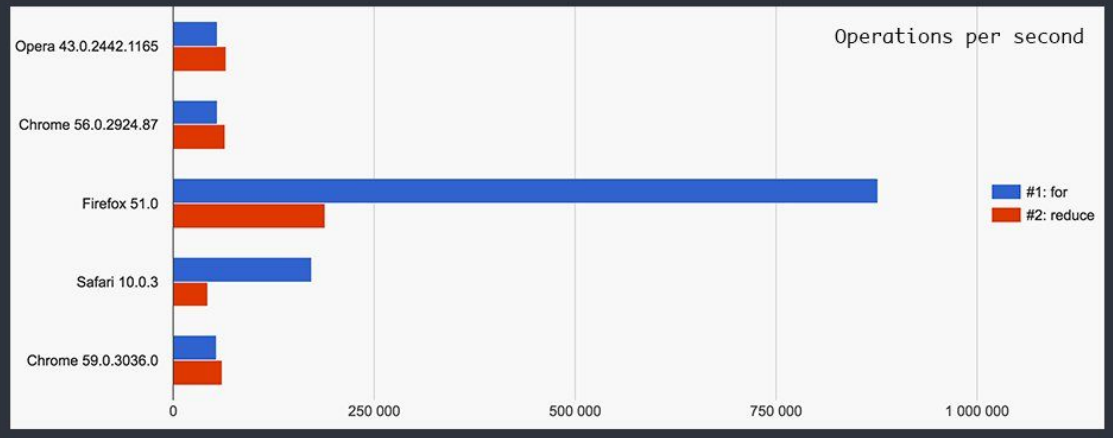
For Loop

```
for(let i = 0; i < arr.length; i++) {
  total += arr[i];
}
```

Array.prototype.reduce

```
total = arr.reduce((a, b) => a + b, 0);
```

Benchmark results (higher is better)



Source: twitter.com/goodmodule/status/840227134689935362

Baseline Performance

Baseline Performance - Builtins

Shubheksha 👩‍💻
@ScribblingOn

Follow

TIL I learnt a simple for loop is faster than forEach in JS, thanks to @TheLarkInn. Now trying to find out why.

Aaron Powell
@slace

Follow

@jdalton watched the video. Reasons make sense. I find it strange that [].foreach is slower than while loops (my tests agree, still strange)

Jani Tarvainen
@velmu

Follow

"[Node.js 6.0] native Promise performance is still around six times slower than Bluebird" blog.wikimedia.org/2017/02/17/nod … /cc @PetkaAntonov #nodejs

Sean Larkin
@TheLarkInn

Following

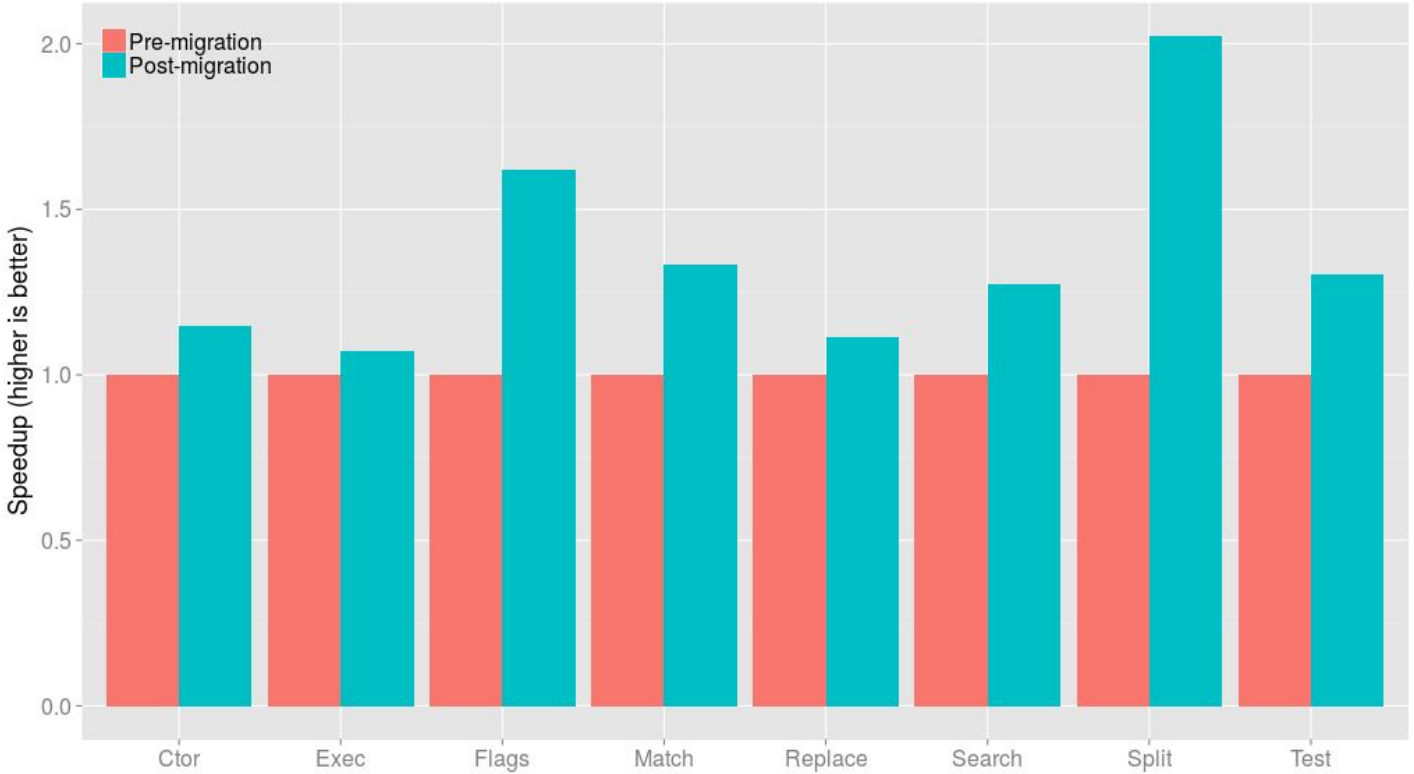
Oo I have some fun low hanging perf fruit for somone looking to make #webpack build faster. Change forEach to for() with let 😍😍

vmb @vsemozhetbyt
@bmeurer @TheLarkInn @ScribblingOn Seems v8 in @bmeurer's bench has let/const cycle optimization. Gist/results (var) gist.github.com/vsemozhetbyt/7…

Baseline Performance - Builtins

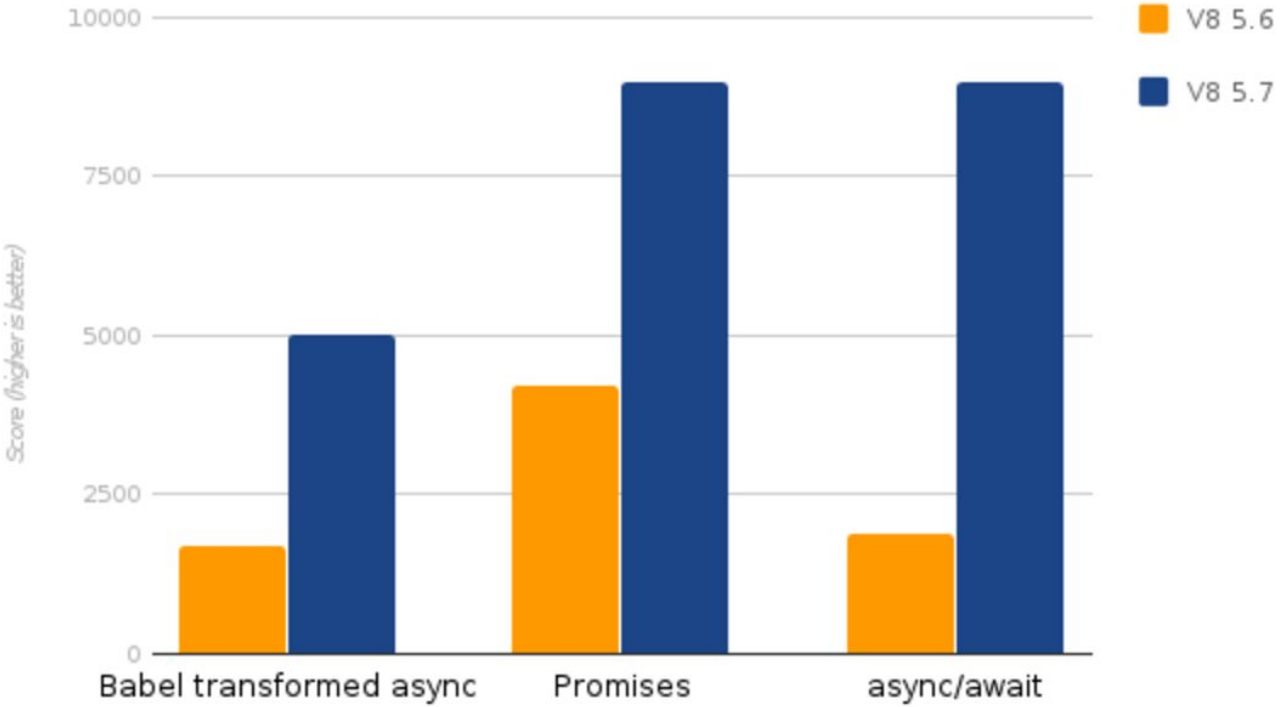
- Expectation mismatch: “builtins should be fast”
- TurboFan’s CodeStubAssembler basis for fast builtins
- Ideally no performance cliff

Baseline Performance - RegExp Builtins



Source: v8project.blogspot.com/2017/01/speeding-up-v8-regular-expressions.html

Baseline Performance - Promises & async/await

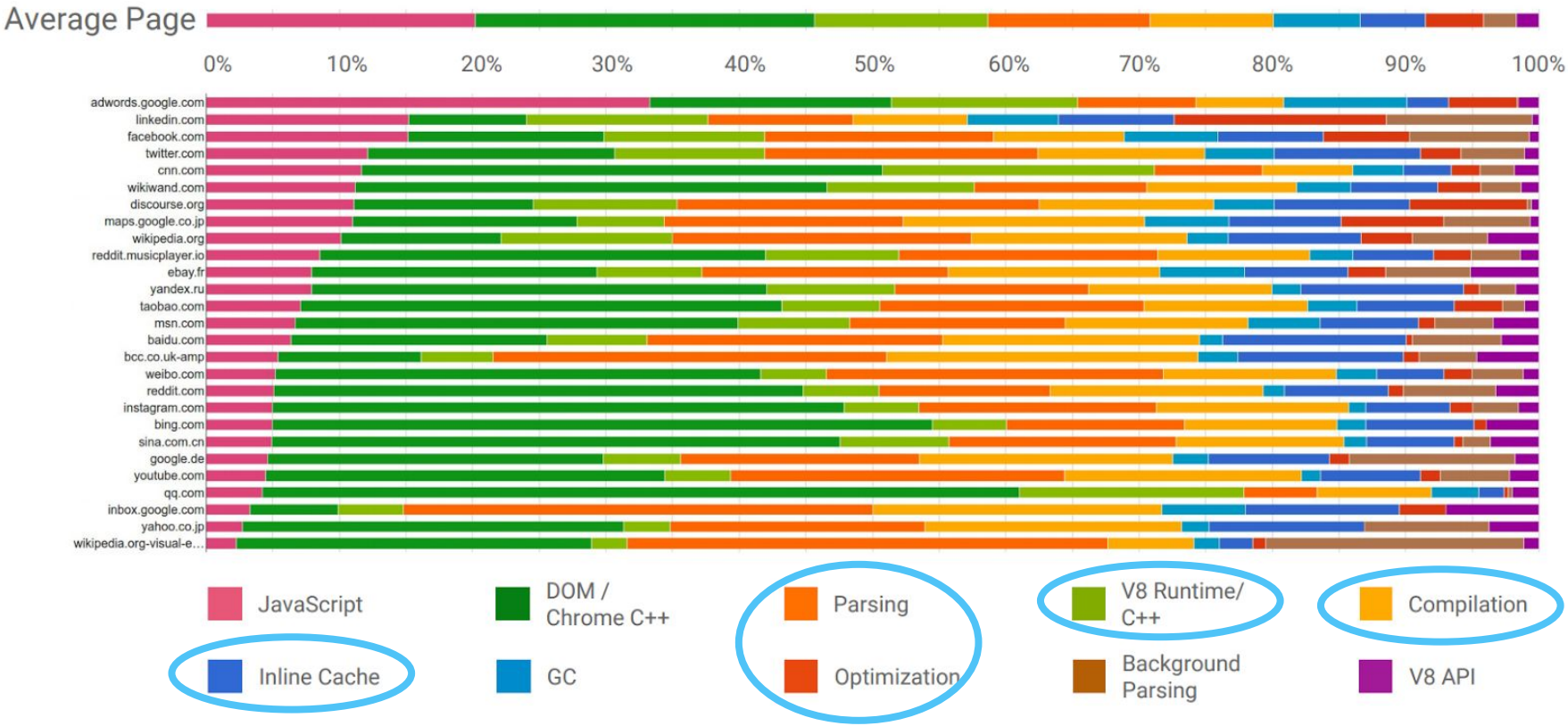


Source: v8project.blogspot.com/2017/02/v8-release-57.html

Baseline Performance

- Baseline performance matters
- Optimize too early is costly
  - Might deoptimize quickly because type feedback not stable yet
  - Optimization is expensive
- Generated code quality depends on quality and stability of type feedback

Improve Startup Time


Average Page
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
adwords.google.com
linkedin.com
facebook.com
twitter.com
cnn.com
wikiwand.com
discourse.org
maps.google.co.jp
wikipedia.org
reddit.musicplayer.io
ebay.fr
yandex.ru
taobao.com
msn.com
baidu.com
bcc.co.uk-amp
weibo.com
reddit.com
instagram.com
bing.com
sina.com.cn
google.de
youtube.com
qq.com
inbox.google.com
yahoo.co.jp
wikipedia.org-visual-e...
JavaScript
DOM / Chrome C++
Parsing
V8 Runtime/ C++
Compilation
Inline Cache
GC
Optimization
Background Parsing
V8 API

Improve Startup Time

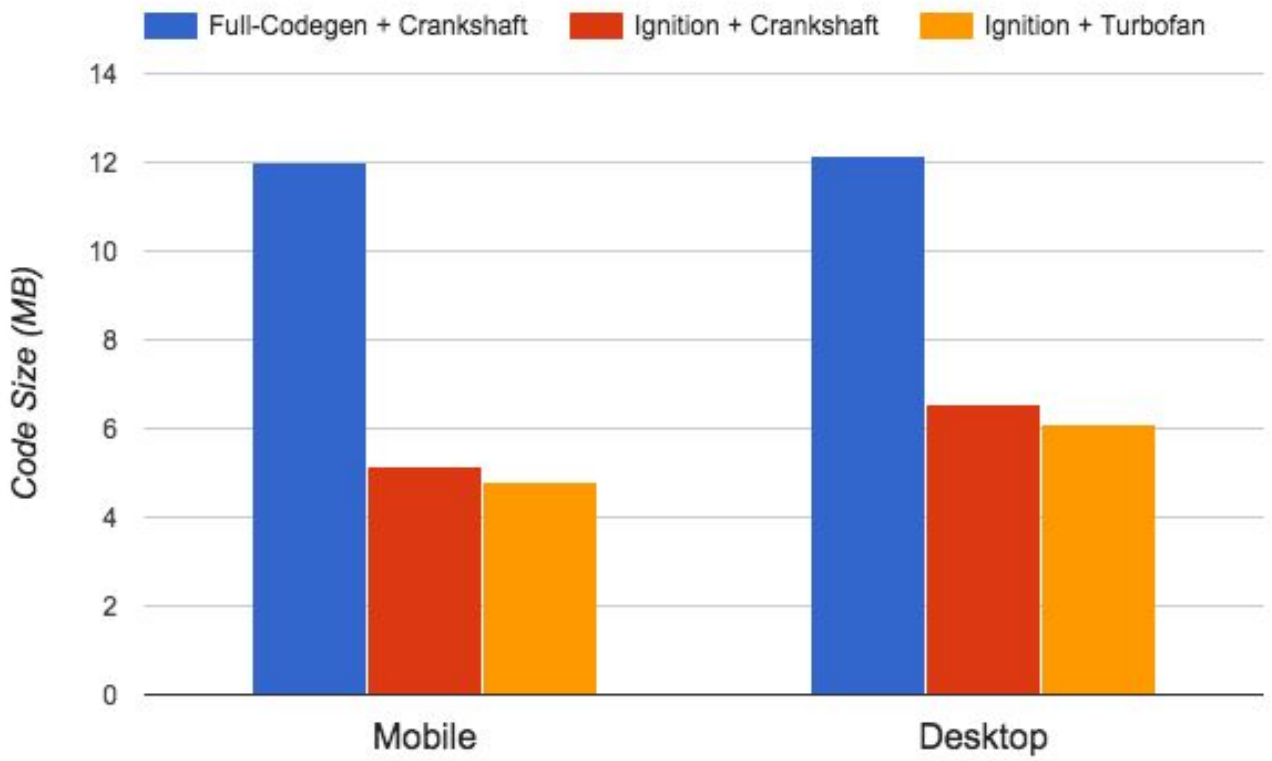
- Bytecode faster to generate
- Better suited for smaller icache - (low-end) mobile
- Parse only once, optimize from bytecode
- Optimize less aggressively - better baseline performance
- Data-driven ICs reduce slow path cost

Reduce Memory Usage

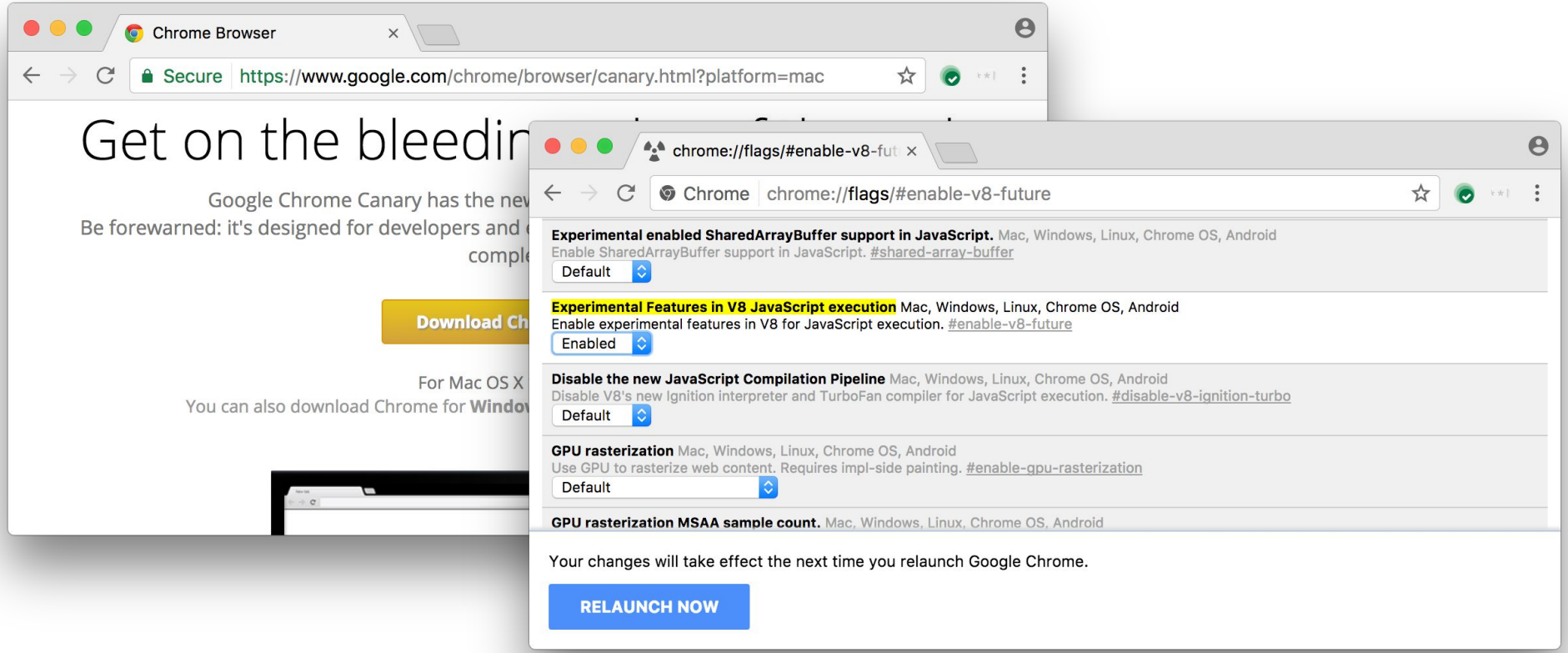
Reduce Memory Usage

- Memory is precious on mobile
- Android devices with 512MiB - 1GiB are common
- Emerging markets
- Ignition code is up to **8x** smaller than Full-Codegen code (arm64)

Reduce Memory Usage



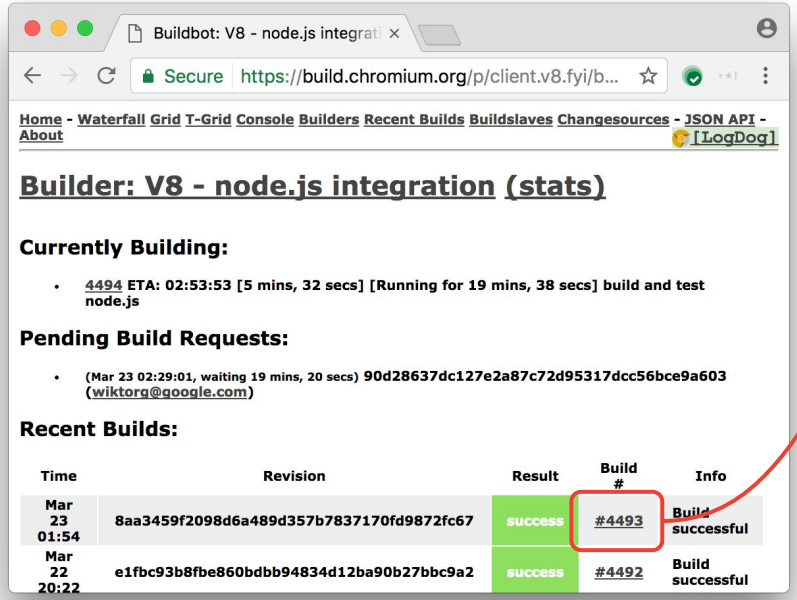
Chrome Dev / Canary

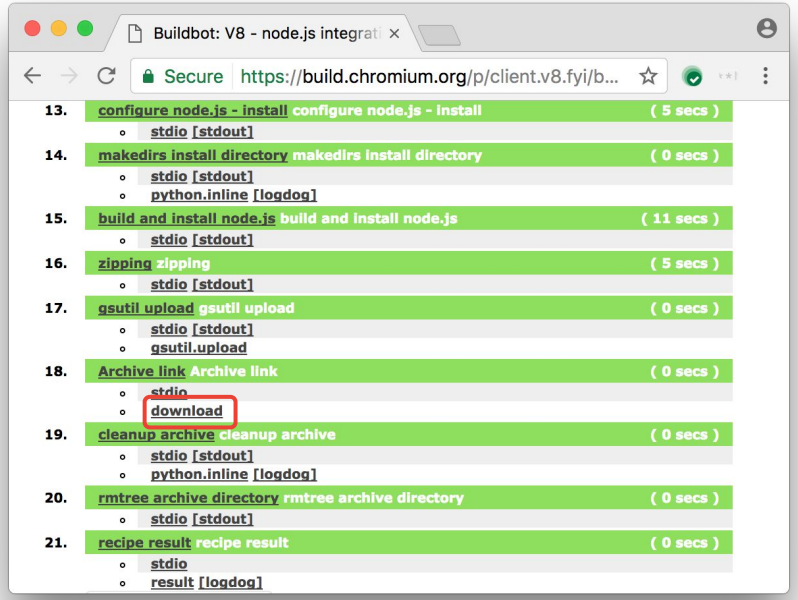


Further information: v8project.blogspot.com/2017/02/help-us-test-future-of-v8.html

Node vee-eight-lkgr (Ubuntu/x86_64 build)

github.com/v8/node





Further information: medium.com/@bmeurer/help-us-test-the-future-of-node-js-6079900566f

Node vee-eight-lkgr (source build)

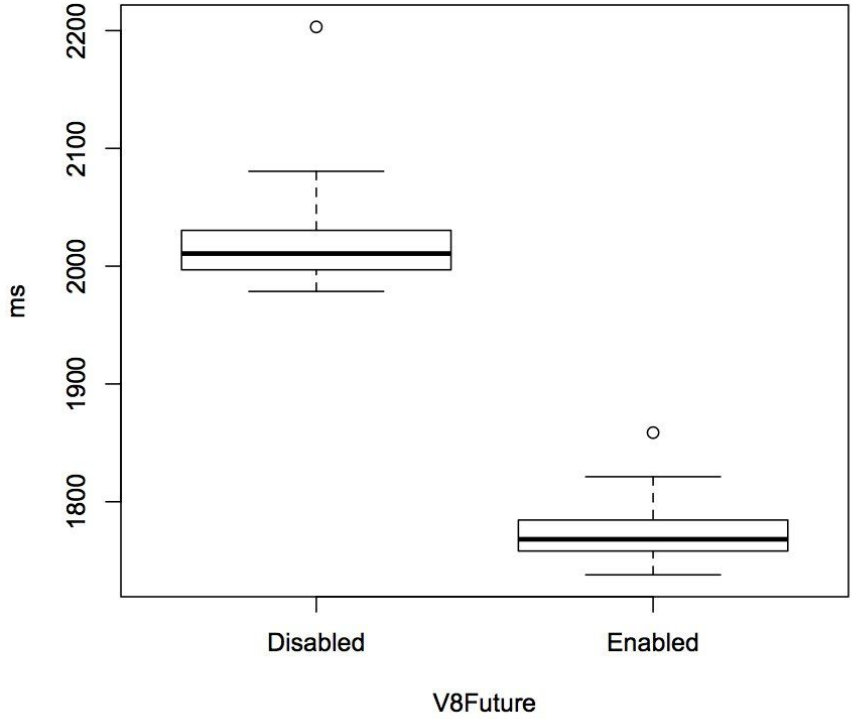
```
git clone https://github.com/v8/node.git node-v8
cd node-v8
git checkout vee-eight-lkgr
./configure --prefix "$HOME/Applications/node-vee-eight-lkgr"
make install
```


Further information: medium.com/@bmeurer/help-us-test-the-future-of-node-js-6079900566f

Performance Results

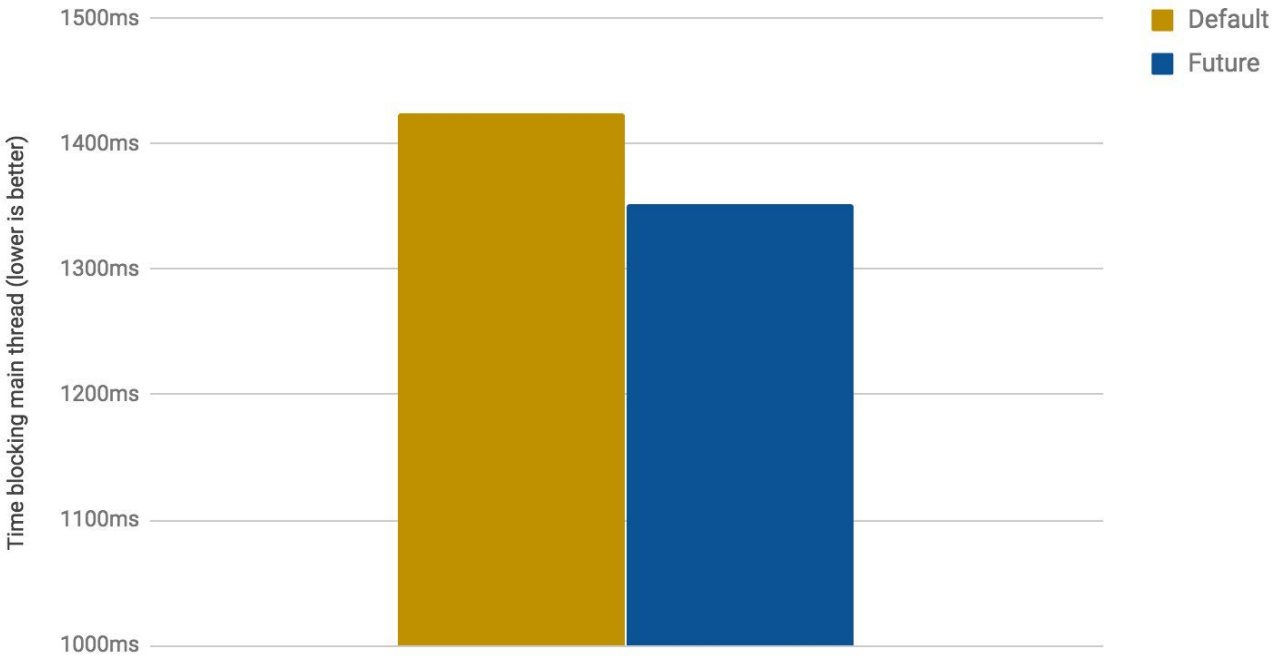
Page Load Time (Top25 Websites)

Page Load Time (LinkedIn Feed)



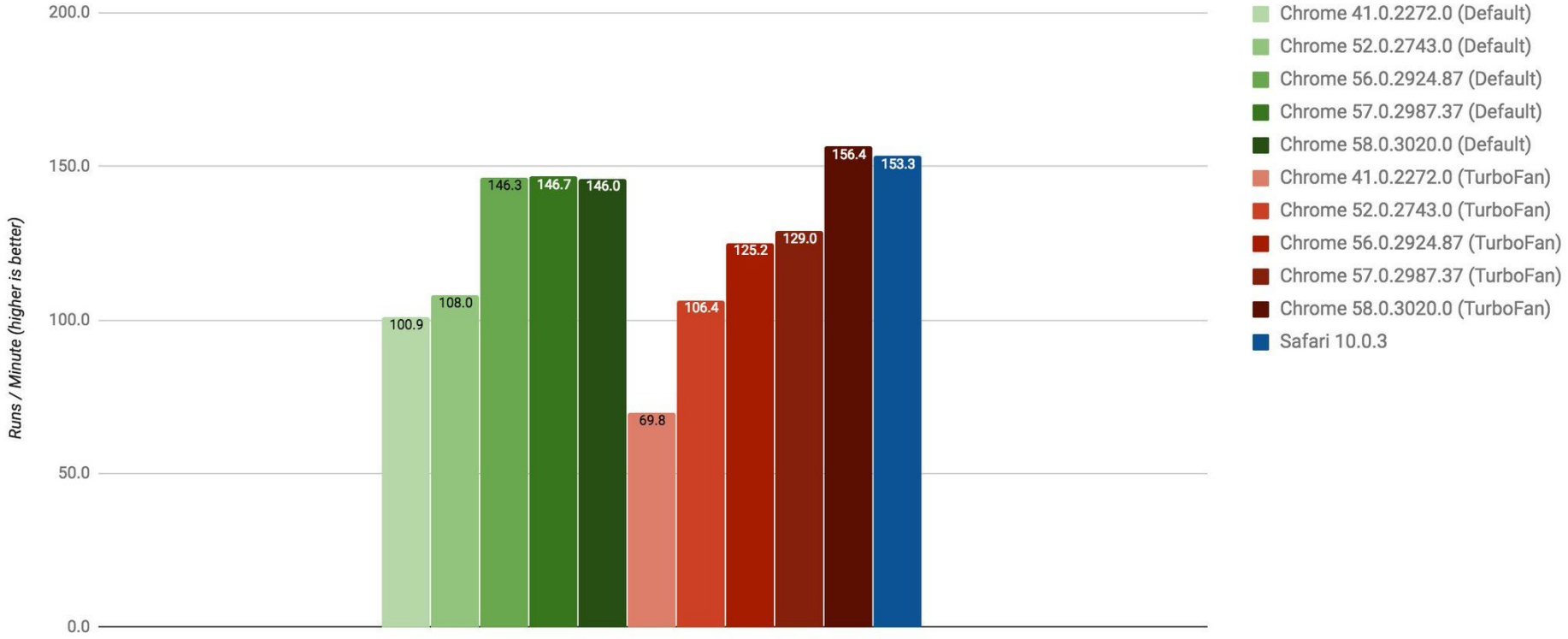
Source: twitter.com/krisselden/status/839886186218823681

Page Load Time (Script Execution)



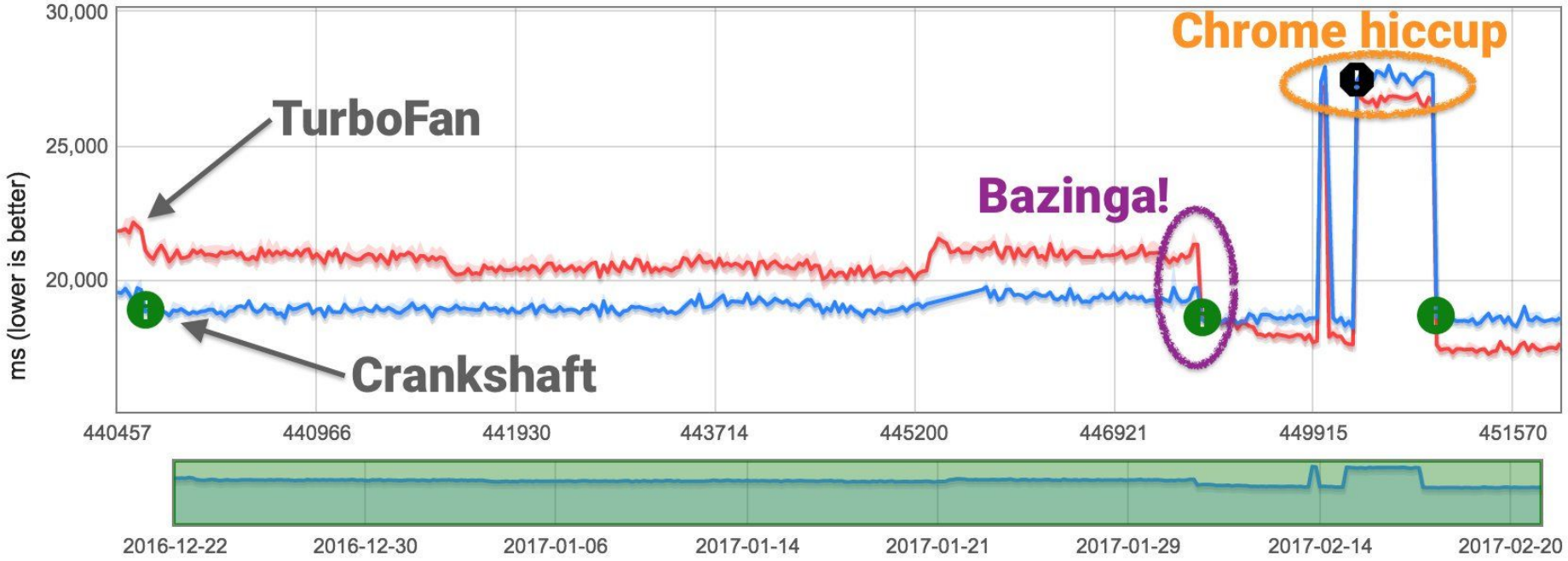
Time HTML parser blocked on script execution

Framework Performance (Speedometer)



Source: browserbench.org/Speedometer

Framework Performance (Speedometer)



Source: browserbench.org/Speedometer

Framework Performance (Ember)

HTML Text

Render Complex List (HTML)

Name	Speed	Error	Samples	Mean
Render Complex List (HTML) 2.10.0	45.93 / sec	∓ 31.84%	434	21.77 ms

Render Complex List

Name	Speed	Error	Samples	Mean
Render Complex List 2.10.0	49.07 / sec	∓ 27.19%	452	20.38 ms

HTML Text

Render Complex List (HTML)

Name	Speed	Error	Samples	Mean
Render Complex List (HTML) 2.10.0	25.89 / sec	∓ 136.14%	267	38.63 ms

Render Complex List

Name	Speed	Error	Samples	Mean
Render Complex List 2.10.0	27.18 / sec	∓ 124.19%	281	36.80 ms

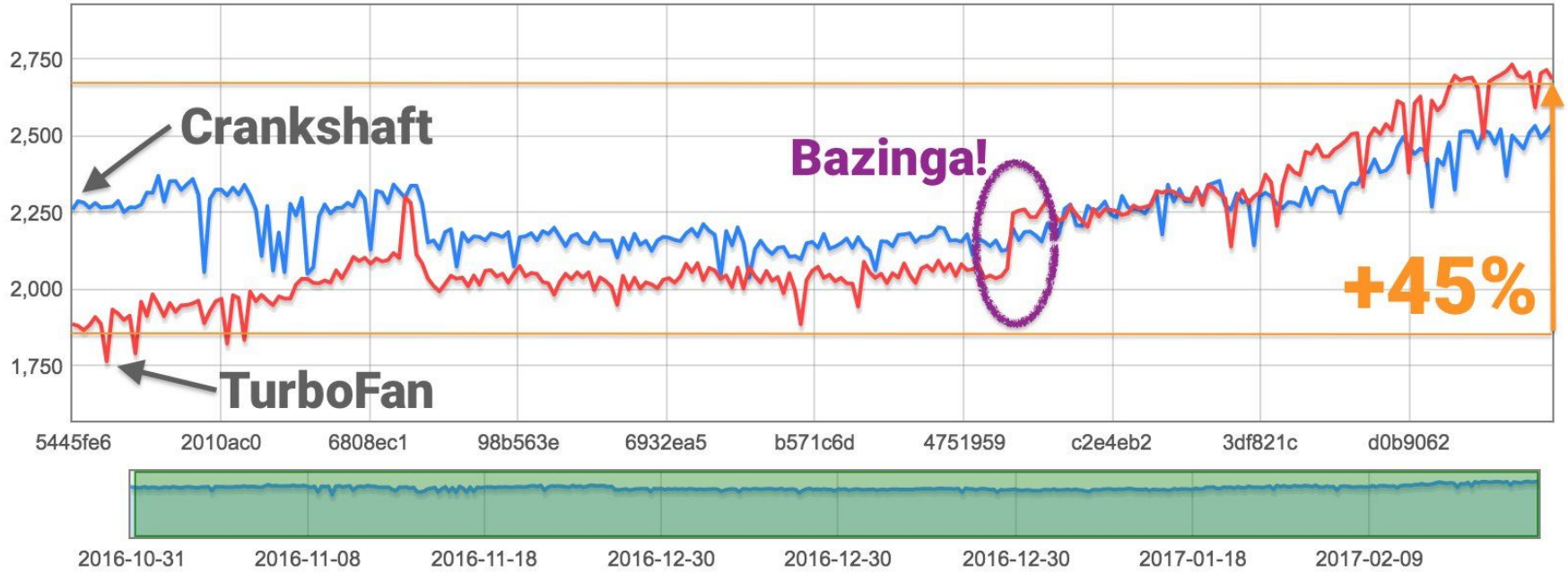
Jeff Atwood ✓
@codinghorror

Following

Seeing some fantastic complex JS perf improvements (Discourse, Ember) in Chrome Canary!! cc @bmeurer

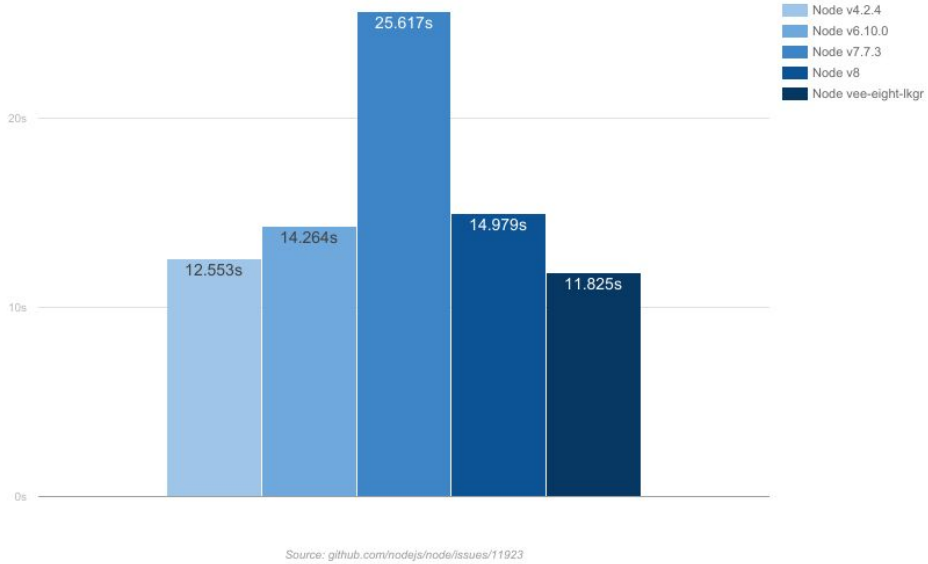
Source: twitter.com/codinghorror/status/841775015917887488

Node Server Workloads (AcmeAir)



Throughput on AcmeAir

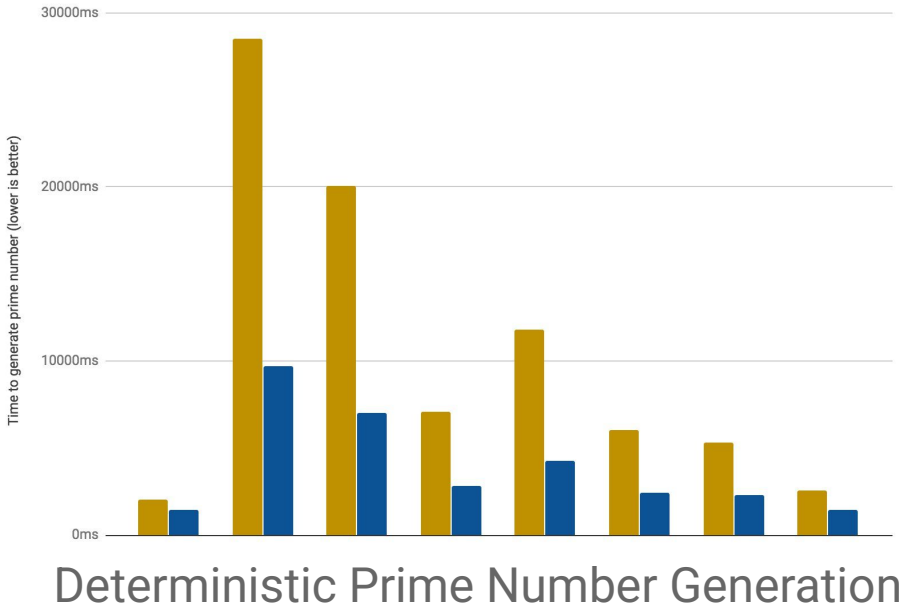
UglifyJS2



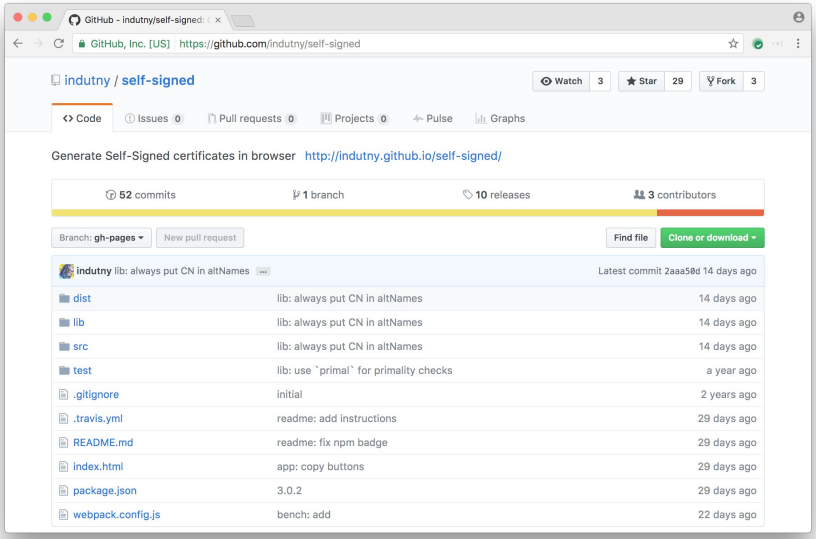
UglifyJS is a JavaScript parser, minifier, compressor or beautifier toolkit.

github.com/mishoo/UglifyJS2

Cryptographic / Number Crunching Applications



Deterministic Prime Number Generation

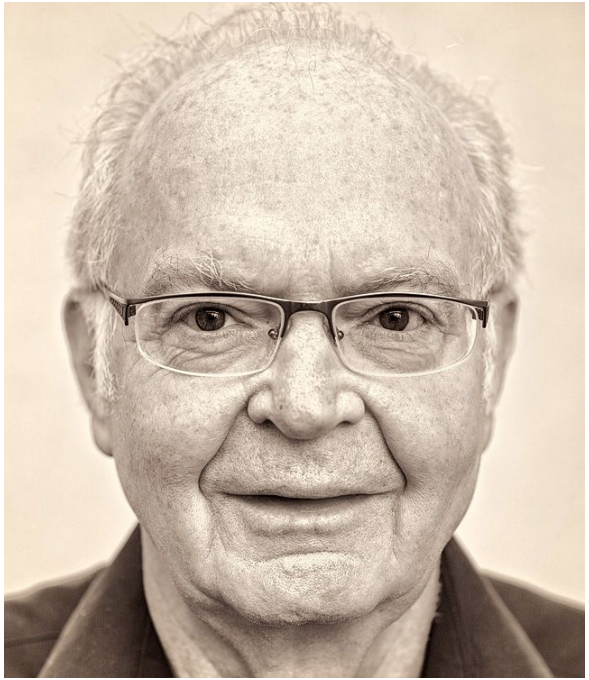


Source: github.com/indutny/self-signed

Performance Advice

Performance Advice

“Premature optimization is the root of all evil”



Performance Advice

- Write idiomatic, declarative JavaScript
  - Appropriate language features
  - Handle exceptions where necessary
  - Use proper collections (Map, Set, WeakMap, WeakSet, etc.)
- Avoid (engine-specific) work-arounds
- File bug reports when something is slow

Performance Advice - Declarative JavaScript

Declarative

```
if (obj !== undefined) { return obj.x; }
```

```
... ...........
23 cmpq [r13-0x60],rax
27 jz 72
... ...........
```

Obscure

```
if (obj) { return obj.x; }
```

```
... ...........
 27 cmpq [r13-0x40],rax
 31 jz 128
 37 test al,0x1
 39 setzl bl
 42 movzxbl rbx,rbx
 45 cmpl rbx,0x0
 48 jnz 185
 54 cmpq [r13-0x38],rax
 58 jz 128
 64 movq rdx,[rax-0x1]
 68 testb [rdx+0xc],0x10
 72 jnz 128
 78 cmpq [r13+0x50],rdx
 82 jz 160
... ...........
160 vmovsd xmm0,[rax+0x7]
165 movq [rbp-0x18],rbx
169 vxorpd xmm1,xmm1,xmm1
173 vucomisd xmm1,xmm0
177 jz 128
179 movq rbx,[rbp-0x18]
183 jmp 88
185 movq [rbp-0x18],rbx
189 cmpq rax,0x0
193 jz 128
195 movq rbx,[rbp-0x18]
199 jmp 88
... ...........
```

~15% faster on avg!

Performance Advice - Declarative JavaScript

Declarative

```
function foo6(f, ...args) {
  return f(...args);
}
```

Obscure

```
function foo5(f) {
  switch (arguments.length) {
    case 1: return f();
    case 2: return f(arguments[1]);
    case 3: return f(arguments[1], arguments[2]);
    default: {
      var args = [];
      for (var i = 1; i < arguments.length; ++i) {
        args[i - 1] = arguments[i];
      }
      return f.apply(undefined, args);
    }
  }
}
```

~4.5x faster on avg!