Google

An overview of the TurboFan compiler

Benedikt Meurer / bmeurer@chromium.org / @bmeurer

V8 Compiler Tech Lead

Google Munich

Overview

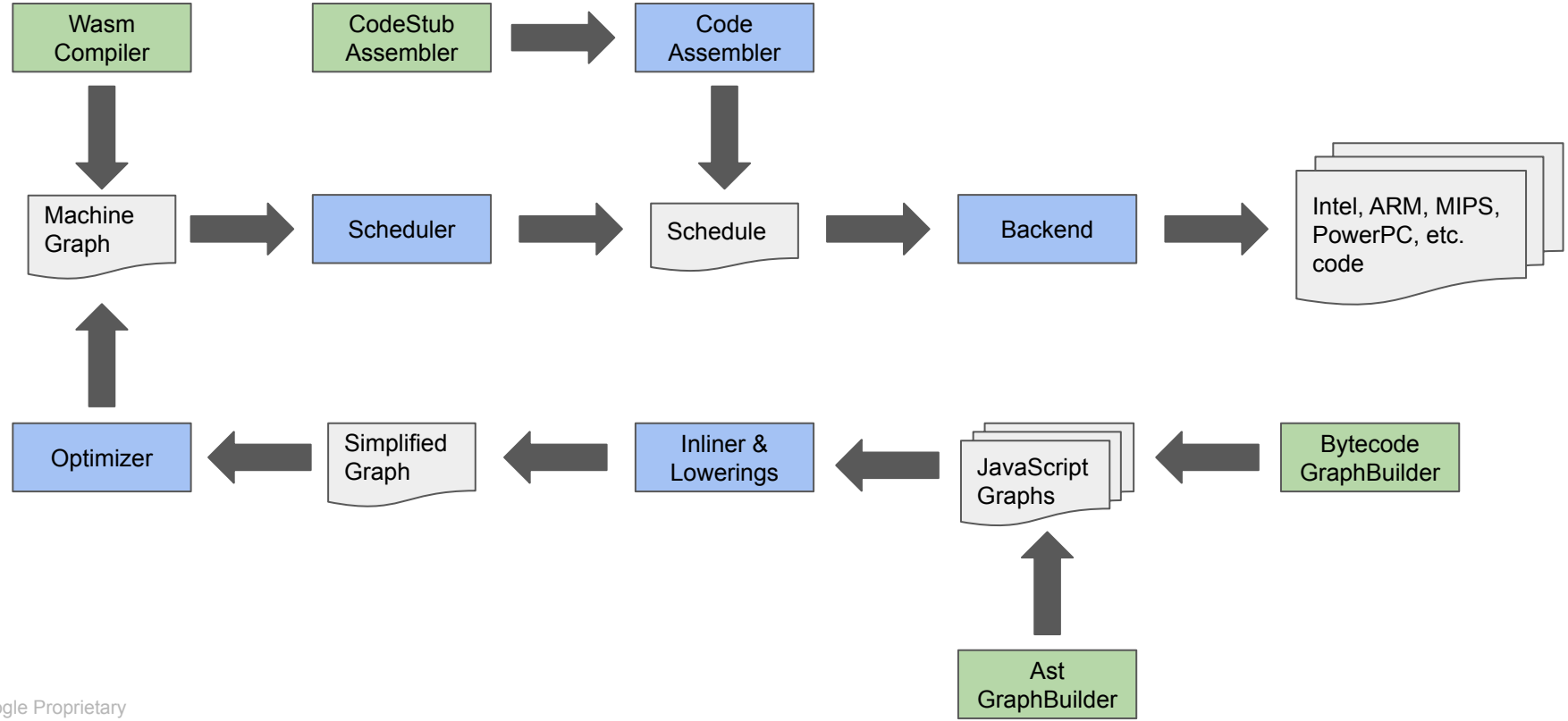
What is TurboFan?

- **the new** JavaScript compiler to replace the aging Crankshaft
- a high-level, portable assembler

Why a portable assembler?

- Interpreter bytecode handlers
- Code stubs and builtins
- asm.js & WebAssembly

The TurboFan architecture / entry points



TurboFan as a portable assembler

Motivation

Traditionally V8 contains a lot of hand-written native code:

- Fast paths for EcmaScript builtins (i.e. `Object.prototype.toString`)
- IC handlers and dispatchers
- Code stubs (i.e. `FastNewClosureStub`)

Native code on the web efforts:

- asm.js
- WebAssembly

Goals

- Provide a uniform code generation architecture
- Reduce porting / maintenance overhead of V8 (currently already 10 ports!)
- Remove performance cliffs due to slow builtins
- Make experimenting with new features easier (i.e. changes to load/store ICs, bootstrapping an interpreter)

CodeAssembler vs. CodeStubAssembler

- CodeAssembler as shim to the TurboFan machine-level IR; wraps RawMachineAssembler, which generates a Schedule with machine instructions
- CodeStubAssembler adds high-level V8/JavaScript-related macros on-top
- InterpreterAssembler adds Ignition-related macros on-top of CodeStubAssembler
- CodeAssembler is portable assembler entry point for TurboFan

Benefits

- Fewer performance cliffs
- Better baseline performance (real world benefit, i.e. `ForInFilter`, `Object.prototype.toString`, `Function.prototype.bind`, etc.)
- Fewer bugs (no more manual register allocation, etc.)
- Reduce porting overhead
- Refactoring viable

TurboFan as JavaScript compiler

Motivation

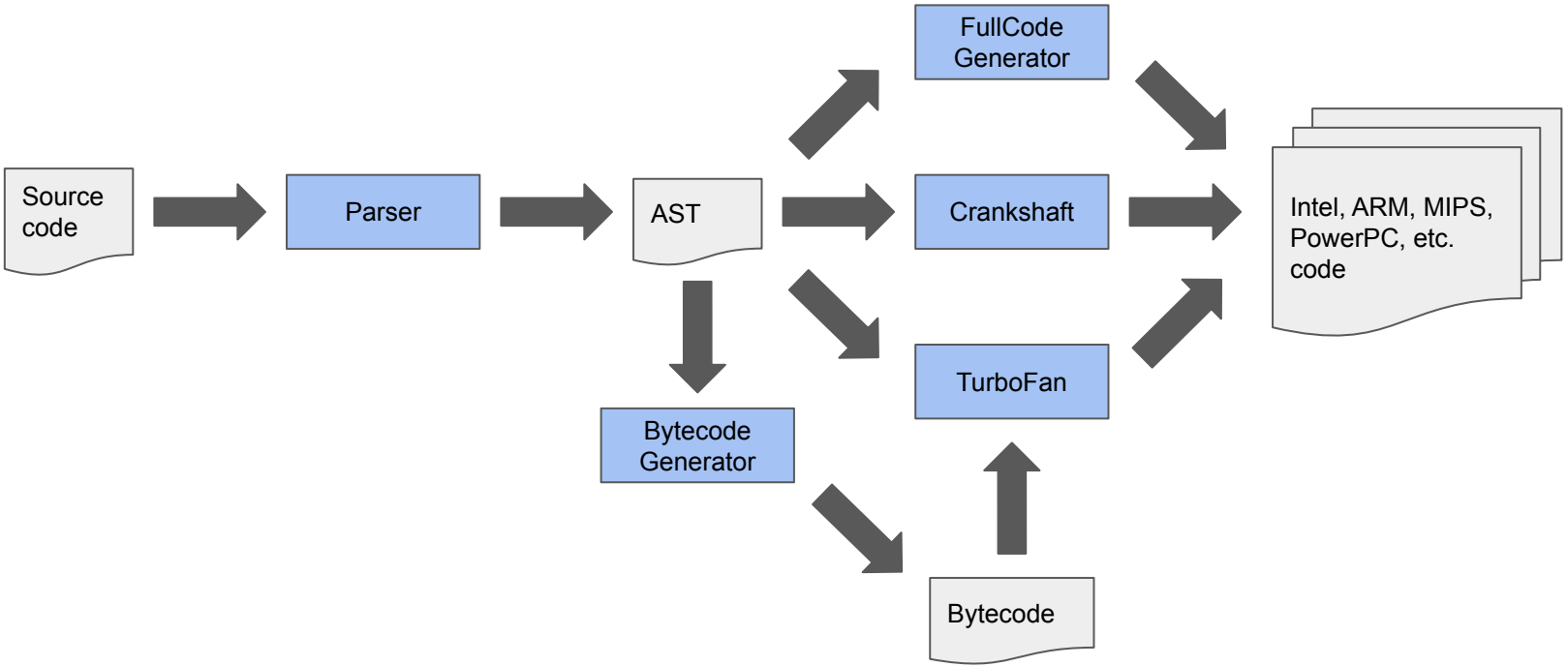
Crankshaft served us well, but has various shortcomings:

- Doesn’t scale to full, modern JavaScript (`try-catch`, `for-of`, generators, `async/await`, …)
- Defaults to deoptimization (performance cliffs, deoptimization loops)
- Graph construction, inlining and optimization all mixed up
- Tight coupling to fullcodegen / brittle environment tracking
- Limited optimization potential / limited static analysis (i.e. type propagation)
- High porting overhead
- Mixed low-level and high-level semantics of instructions

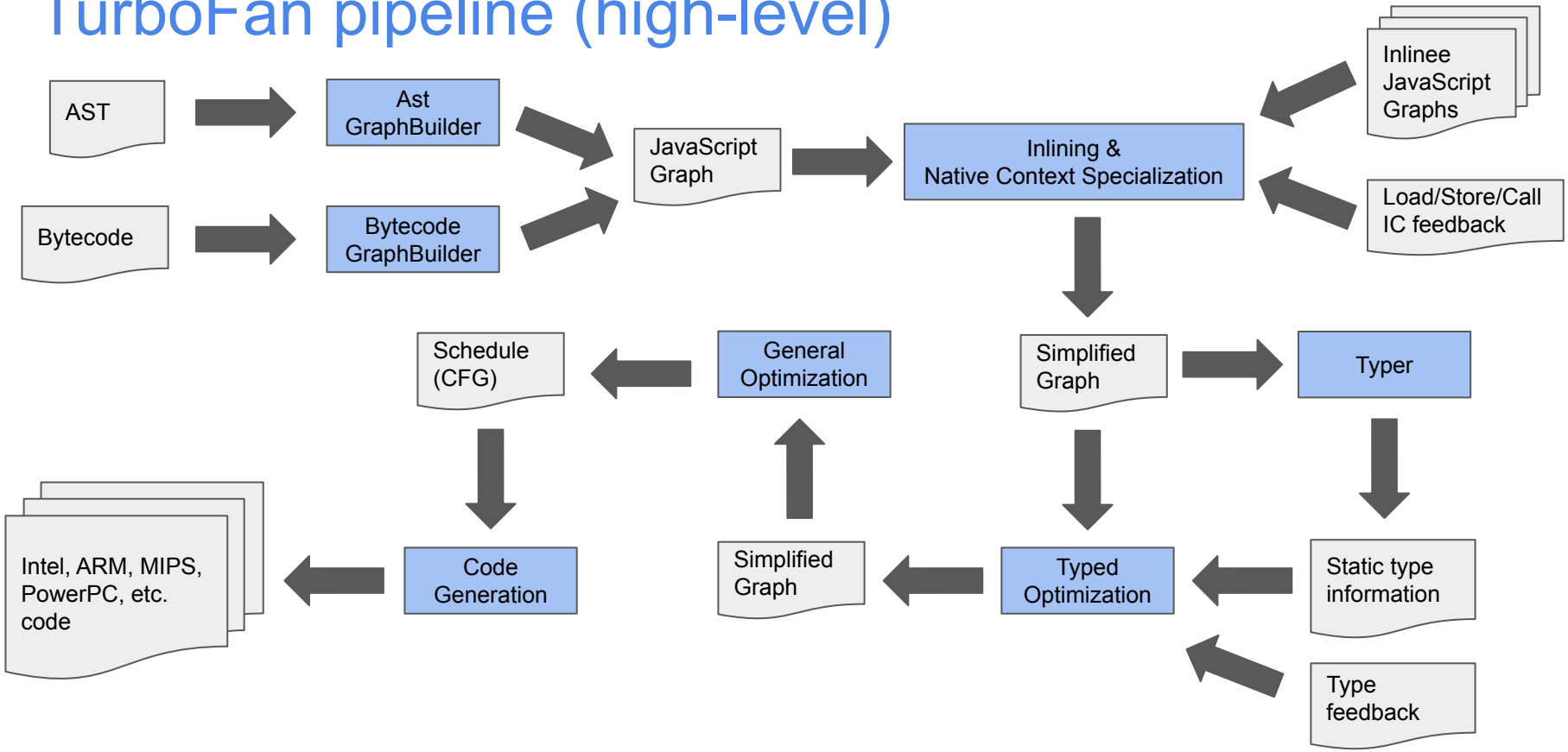
Goals

- Full ESnext language support (`try-catch/-finally`, `class` literals, `eval`, generators, `async` functions, modules, destructuring, etc.)
- Utilize and propagate (static) type information
- Separate graph building from optimization / inlining
- No deoptimization loops / deoptimization only when really beneficial
- Sane environment tracking (also for lazy deoptimization)
- Predictable peak performance

V8 compilation overview



TurboFan pipeline (high-level)



Graph Building

- Turns Bytecode or AST into graph with JavaScript nodes
  - JSAdd, JSCallFunction, JSLoadProperty, etc.
  - Branch, IfTrue, IfFalse, etc.
- Tracks environment for baseline code and inserts appropriate FrameStates
  - CheckPoint with FrameState for eager deoptimization
  - FrameState after nodes that can lead to arbitrary JS execution (i.e. everything except JSTypeOf, JSStrictEqual and JSToBoolean) for lazy deoptimization
- Eliminates dead FrameState uses, i.e. values that baseline cannot see are replaced with optimized_out sentinel
- Optional OSR deconstruction right afterwards to remove the non-OSR entry

Native Context Specialization & Inlining

- Specialize global object property access to the global object for native context
- Specialize to native context based on Load/Store/Call IC feedback
  - JSLoadNamed[x](o) turns into fast access for x on o if baseline collected feedback
  - Specialize function calls to direct calls (i.e. to known JSFunctions) when Call IC provides feedback
- Inline through JSCallFunction and JSCallConstruct sites, even polymorphic sites
  - Supports sane heuristics (not only first come, first serve)
  - Can even inline through `Function.prototype.call`/`.apply`
- Eliminates dead nodes in the (control) graph on-the-fly
- Doesn't take into account type feedback for binary/compare operations

Typer & Typed Optimization

- Typer assigns types to value producing nodes, based on JS typing rules
  - Already considers types for fields known from native context specialization
  - Propagates type information through phis and sigmas (loop exits)
  - Range analysis and constant folding (some) included
- Lowering JS graph to simplified graph based on types
  - Take into account *static* type information first, but also utilize type feedback (binary/compare)
  - Inline builtin calls based on parameter types (i.e. Math.abs, Array.prototype.push, etc.)
- Lower JS object creation nodes to inline allocations
  - JSCreate becomes an inline allocation when new.target is known
  - Similar for JSCreateArray, JSCreateLiteralObject, etc.
  - Makes those eligible for escape analysis and load/store/check elimination

General Optimizations

- Loop peeling
  - No more deoptimization loops because of aggressive hoisting
- Load/check elimination
  - Kills redundant loads and checks
  - Includes global value numbering pass
- Escape analysis
  - Eliminate non escaping allocations
  - Scalar replacement of aggregates
  - Not yet partial; allocation sinking planned
- Representation selection
  - Truncation analysis
  - Type feedback propagation

General Optimizations

- Effect/control linearization
  - Wires potentially deoptimizing nodes into effect/control chain, assigns check points
  - Expands macro operations with internal control flow (i.e. ChangeTaggedToFloat64)
  - Optimizes branches that depend on phis (branch cloning)
- Redundant store elimination
- Control flow optimization (turns certain branch chains into switches)
- Allocation folding and write barrier elimination
  - Fragmentation-free folding of allocations
  - Even across branches; not yet across merges
- Late optimization pass
  - Another global value numbering pass
  - Strength reduction, dead code elimination, redundant branch elimination

Code Generation

- Sophisticated instruction selection
  - Pattern matching on subgraphs for often optimal cover
  - Architecture-dependent matchings
  - Dead instruction elimination
- Register allocation
  - Essentially improved version of the lithium allocator
- Jump threading
  - Remove jumps between blocks
- Code emission

Code generation example

Generate code for `add`

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

Run with `--turbo --ignition-staging`

```
add(1,2);
add(2,3);
%OptimizeFunctionOnNextCall(add);
add(3,4);
```

Almost optimal code sequence already (Smi representation missing)

Code generation example

```
                -- B0 start (construct frame) --
0x3caafc104060     0  55             push rbp
0x3caafc104061     1  4889e5         REX.W movq rbp,rsp
0x3caafc104064     4  56             push rsi
0x3caafc104065     5  57             push rdi
0x3caafc104066     6  493ba5600c0000 REX.W cmpq rsp,[r13+0xc60]
0x3caafc10406d    13  0f863d000000   jna 80  (0x3caafc1040b0)
                -- B2 start --
                -- B3 start (deconstruct frame) --
0x3caafc104073    19  488b4518       REX.W movq rax,[rbp+0x18]
0x3caafc104077    23  a801           test al,0x1
0x3caafc104079    25  0f8548000000   jnz 103  (0x3caafc1040c7)
0x3caafc10407f    31  488b5d10       REX.W movq rbx,[rbp+0x10]
0x3caafc104083    35  f6c301         testb rbx,0x1
0x3caafc104086    38  0f8540000000   jnz 108  (0x3caafc1040cc)
0x3caafc10408c    44  488bd3         REX.W movq rdx,rbx
0x3caafc10408f    47  48c1ea20       REX.W shrq rdx, 32
0x3caafc104093    51  488bc8         REX.W movq rcx,rax
0x3caafc104096    54  48c1e920       REX.W shrq rcx, 32
0x3caafc10409a    58  03d1           addl rdx,rcx
0x3caafc10409c    60  0f802f000000   jo 113  (0x3caafc1040d1)
0x3caafc1040a2    66  48c1e220       REX.W shlq rdx, 32
0x3caafc1040a6    70  488bc2         REX.W movq rax,rdx
0x3caafc1040a9    73  488be5         REX.W movq rsp,rbp
0x3caafc1040ac    76  5d             pop rbp
0x3caafc1040ad    77  c21800         ret 0x18
                -- B4 start (no frame) --
                -- B1 start (deferred) --
```

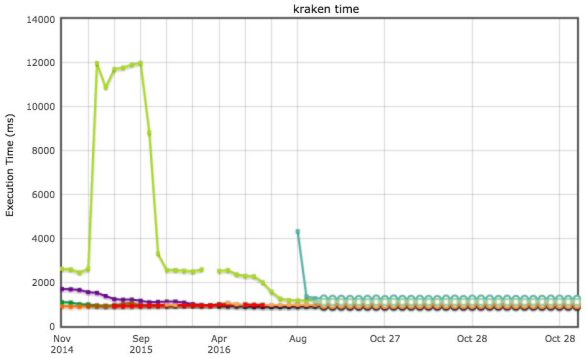
```
0x3caafc1040b0    80  48bbd012abb3397f0000 REX.W movq rbx,0x7f39b3ab12d0
0x3caafc1040ba    90  33c0           xorl rax,rax
0x3caafc1040bc    92  488b75f8       REX.W movq rsi,[rbp-0x8]
                  -- real.js:229:12 --
0x3caafc1040c0    96  e87b02f0ff     call 0x3caafc004340     ;; code: STUB
0x3caafc1040c5   101  ebac           jmp 19  (0x3caafc104073)
0x3caafc1040c7   103  e834ffd7ff     call 0x3caafbe84000     ;; deoptimization
bailout 0
0x3caafc1040cc   108  e839ffd7ff     call 0x3caafbe8400a     ;; deoptimization
bailout 1
0x3caafc1040d1   113  e83effd7ff     call 0x3caafbe84014     ;; deoptimization
bailout 2
0x3caafc1040d6   118  90             nop
0x3caafc1040d7   119  90             nop
0x3caafc1040d8   120  90             nop
0x3caafc1040d9   121  90             nop
0x3caafc1040da   122  90             nop
0x3caafc1040db   123  90             nop
0x3caafc1040dc   124  90             nop
0x3caafc1040dd   125  90             nop
0x3caafc1040de   126  90             nop
0x3caafc1040df   127  90             nop
0x3caafc1040e0   128  90             nop
0x3caafc1040e1   129  90             nop
0x3caafc1040e2   130  90             nop
0x3caafc1040e3   131  90             nop
                  ;;; Safepoint table.
```

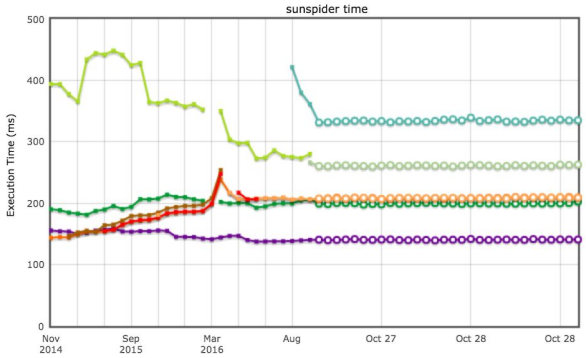
Where do we stand?

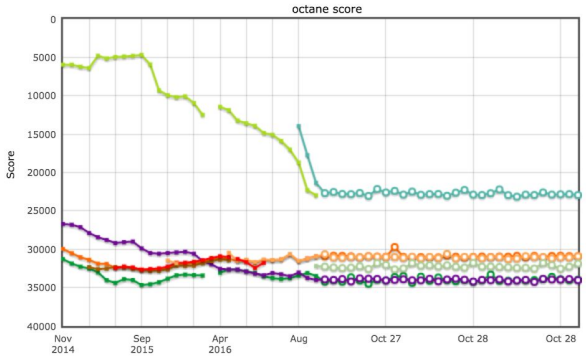
Peak-performance benchmarks

- Chrome (v8)
- Safari (jsc)
- Firefox (Ion)
- Chrome (Turbofan only)
- Firefox (Ion, backtracking)
- Firefox (Ion, unboxed objects)
- Firefox (Ion, testbed regalloc)
- Firefox (non writable jitcode)
- Firefox (flow sensitive aa)
- Firefox (branch pruning)
- Chrome (Turbofan, Ignition)
- Chrome (Ignition)

Machines »
Breakdown »
About »







What’s coming?

Future

- Retire AstGraphBuilder (asm.js only soon)
  - WebAssembly based asm validator coming
- Optimize ESnext features via Ignition+TurboFan
- Add missing optimizations to TurboFan (i.e. arguments object, Smi representation, etc.)
- Eventually route more (traditional) JavaScript through Ignition+TurboFan only

Upcoming talks

- TurboFan IR overview (jarin@chromium.org)
  - Different IR layers (JavaScript, simplified, machine)
  - Sea-of-nodes overview
- TurboFan deoptimization support (jarin@chromium.org)
  - Check point chain
  - Lazy bailout
  - Deoptimizer support
  - Liveness analysis

Questions?