V8: Behind the Scenes (February Edition feat. A tale of TurboFan)

MARCH 1, 2017

February has been an exciting and very, very busy month for me. As you have probably heard, we've finally announced that we will launch the Ignition+TurboFan pipeline in Chrome 59. So despite running late, and not making it for February actually, I'd like to take the time to reflect on the TurboFan tale a bit, and tell my story here. Remember, that everything you read here is my very personal opinion and doesn't reflect the opinion of V8, Chrome or Google.

It's been almost 3½ years since we started (thinking about) TurboFan in 2013. The world has changed a lot since that time. And V8 has changed a lot since then. I have changed a lot since that time. And my mental model of JavaScript, the web and Node.js has changed significantly. This story of TurboFan development is strongly linked to my own personal development, so it's probably heavily biased towards my own point of view here.

In late 2013, when I joined the TurboFan project, we strongly believed that we had to address the code generation problems of Crankshaft and throw more sophisticated peak performance optimizations at JavaScript code. We based most of these findings on JavaScript code we hit in certain benchmarks like Octane and investigations of asm.js based applications, but also on findings from important web pages like Google Maps. Those were considered good proxies for real-world performance, as they stress the optimizing compiler a lot. Looking back, we couldn't have been more wrong. While it's true that various tests in Octane could benefit from an even smarter compiler, the reality was that for the vast majority of websites the optimizing compiler doesn't really matter, and can even hurt the performance —— because speculative optimizations come at a cost —— especially during page load and in particular on mobile devices.

But for the first year of TurboFan development we were mostly unaware of the real-world concerns. Our initial goal was to build a full language optimizing compiler that does extremely well on asm.js like code —— both areas where Crankshaft was never really able to

shine. In Chrome 41 we shipped TurboFan for asm.js code. This initial version of TurboFan already contained a lot of smartness. We basically got to Firefox level of asm.js performance with a more general approach. Most of the type based optimizations for fast arithmetic would work equally well in general JavaScript. From my very personal point of view, the TurboFan optimizing compiler at that time was probably the most beautiful version we ever had, and the only version (of a JavaScript compiler) where I could imagine that a "sea of nodes" approach might make sense (although it was already showing its weakness at that time). In the following months we tried to find incremental ways to turn TurboFan into a viable, general drop-in replacement for Crankshaft. But we struggled to find another subset of JavaScript that would be possible to tackle independently, similar to how we started with asm.js.

In mid 2015 we began to realize that TurboFan might actually solve a problem we don't have, and that we might need to go back to the drawing board to figure out why V8 is struggling in the wild. We weren't really engaging the community at that time, and my personal response when developers brought problems to my attention was often negative and along the lines of "your JavaScript does odd things", which over time turned into "if your code is slow in V8, you wrote slow code" in people's minds. So taking a step back, and trying to understand the full picture, I slowly realized that a lot of the suffering arose from performance cliffs in V8. Phrased differently, we were over-focused on the peak performance case, and baseline performance was a blind spot.

This lack of balance led to highly unpredictable performance. For example, when JavaScript code follows a certain pattern —— avoid all kinds of performance killers, keep everything monomorphic, limit the number of hot functions —— you'll be able to squeeze awesome performance out of V8, easily beating Java performance on similar code. But as soon as you leave this fine line of awesome performance, you often immediately fall off a steep cliff.

V8 has been like this cliff. If you pay attention, then it's stunning and beautiful. But if you don't, and you fall off the cliff, you're screwed. Performance differences of **100x** weren't uncommon in the past. Of these cliffs, the `arguments` object handling in Crankshaft is probably the one which people hit most often and which is the most frustrating too. The fundamental assumption in Crankshaft is that `arguments` object does not escape, and thus Crankshaft does not need to materialize the actual JavaScript `arguments` object **ever**, but instead can just take the parameters from the activation record. So, in other words, there's no safety net. It's all or nothing. Let's consider this simple dispatching logic:

```
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);
  }
```

```
}

for(var i = 0; i < 100000; ++i) {
  dispatch(1, 2, 3, 4, 5);
}
```

Looking at it naively, it seems to follow the rules for the `arguments` object in Crankshaft: In `dispatch` we only use `arguments` together with `Function.prototype.apply`. Yet, running this simple `example.js` in node tells us that all optimizations are disabled for `dispatch`:

```
$ node --trace-opt example.js
...
[marking 0x353f56bcd659 <JS Function dispatch (SharedFunctionI
[compiling method 0x353f56bcd659 <JS Function dispatch (Sharec
[disabled optimization for 0x167a24a58fc9 <SharedFunctionInfo
$
```

The infamous Bad value context for arguments value reason. So, what is the problem here? Despite the code following the rules for `arguments` object, it falls off the performance cliff. The real reason is pretty subtle: Crankshaft can only optimize `fn.apply(receiver,arguments)` if it knows that `fn.apply` is `Function.prototype.apply`, and it only knows that for monomorphic `fn.apply`property accesses. That is, `fn` has to have exactly the same hidden class —— map in V8 terminology —— all the time.
But `callbacks[0]` and `callbacks[1]` have different maps, since `callbacks[0]` is a sloppy mode function, whereas `callbacks[1]` is a strict mode function:

```
$ cat example2.js
var callbacks = [
  function sloppy() {},
  function strict() { "use strict"; }
];
console.log(%HaveSameMap(callbacks[0], callbacks[1]));
$ node --allow-natives-syntax example2.js
false
$
```

TurboFan on the other hand happily optimizes dispatch (using the latest Node.js LKGR):

```
$ node --trace-opt --future example.js
[marking 0x20fa7d04cee9 <JS Function dispatch (SharedFunctionI
[compiling method 0x20fa7d04cee9 <JS Function dispatch (Sharec
[optimizing 0x1c22925834d9 <JS Function dispatch (SharedFuncti
[completed optimizing 0x1c22925834d9 <JS Function dispatch (Sh
...
$
```

In this trivial example, the performance difference is already **2.5x**, and TurboFan doesn't even generate awesome code —— yet. So you take the performance hit just because you're faced with the choice of two extremes: Fast path vs. slow path. And V8's focus on the fast path in the past often led to making the slow path even slower, for example because you pay more for tracking feedback that you need to generate almost perfect code in some fast case, or simply because you have to fall through even more checks to get to the slow path.

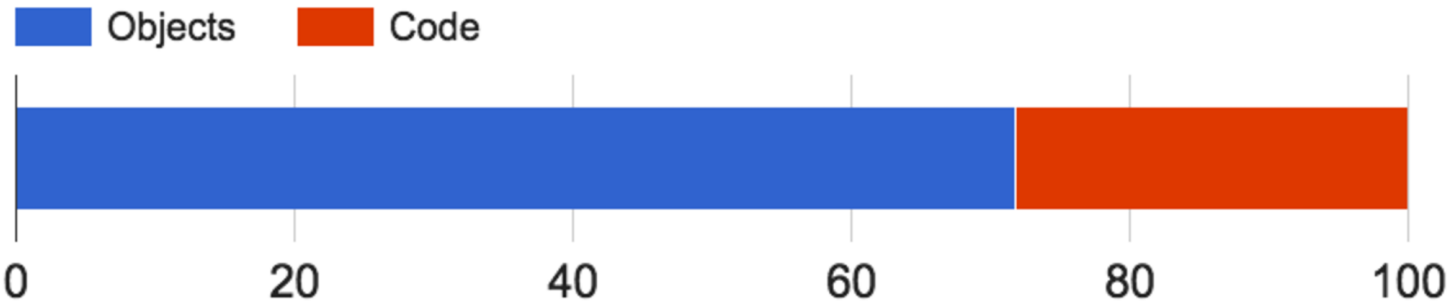
Taking a step back again: If TurboFan was supposed to help us, then it had to do something about the slow path too. And we figured that we'd have to address two things to make this happen:

1. Widen the fast path.
2. Improve the slow path.

Widening the fast path is crucial to ensure that the resources the JavaScript engine spends on trying to optimize your code actually pay off. For example, it's a complete waste of resources to collect type feedback and profile a function until it gets hot, just to then realize that it uses `arguments` in a way that isn't supported. The stated goal for the TurboFan optimizing compiler is to support the full language, and always pay for itself. In the new world, tiering up from Ignition to TurboFan is always a win in terms of execution speed. In this sense, TurboFan is kind of a *better Crankshaft*.

But that alone wouldn't really help, especially since TurboFan compilation is more expensive than Crankshaft (you really have to acknowledge the awesome engineering work that went into Crankshaft here, which still shines as core part of the Dart engine). In fact real-world performance would have suffered a lot from just replacing Crankshaft with TurboFan in many cases. And real-world performance is starting to hurt V8 and Chrome seriously, as we move to a world where most of the web traffic comes from mobile devices,

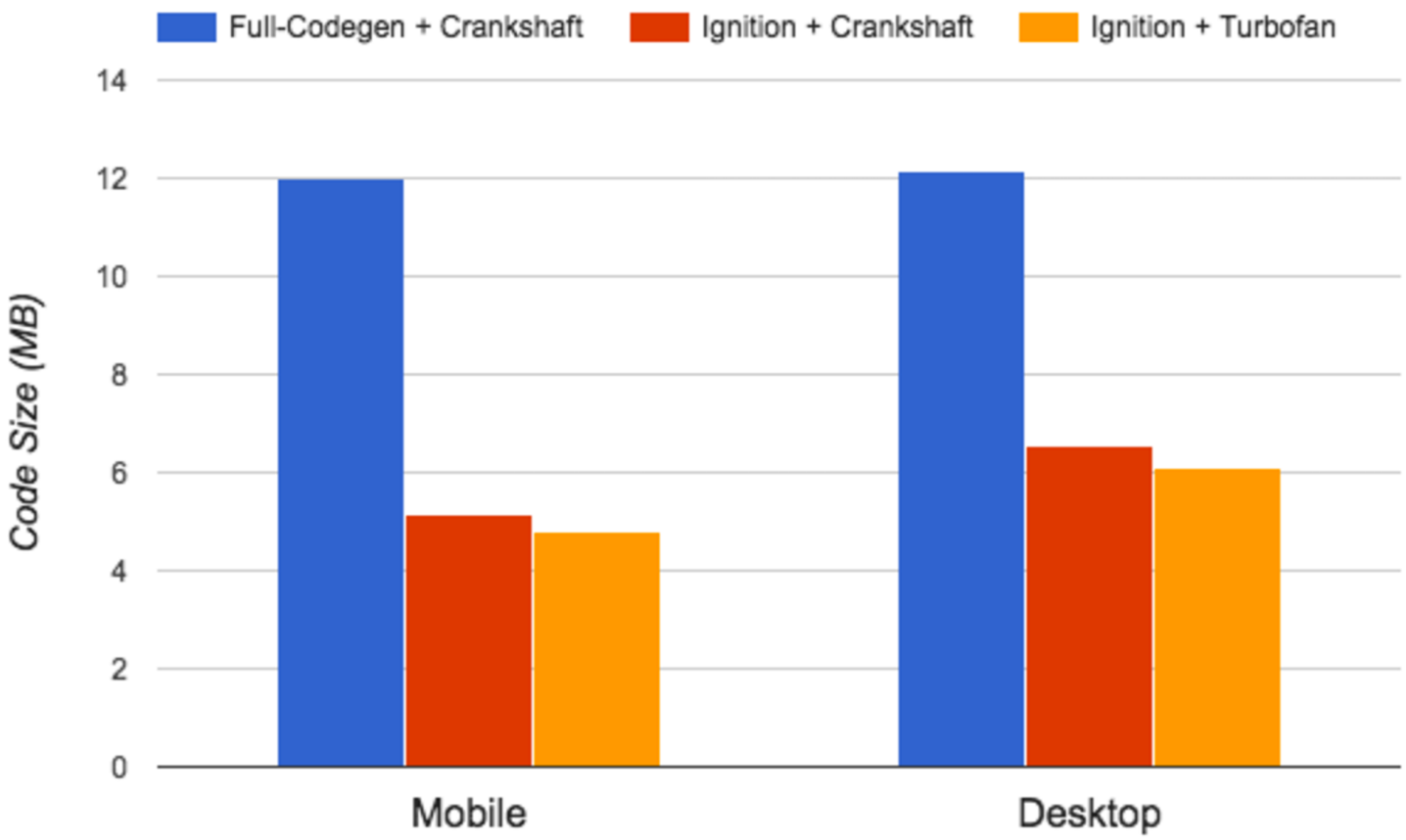
and more and more of these devices are low-end Android devices. In this world, page load time and low overhead —— both memory and execution wise —— are crucial for success. For example, we discovered that **30%** of the managed memory in typical web applications was used by Code objects:



Source: V8: Hooking up the Ignition to the TurboFan, BlinkOn 7 conference, @rossmcilroy and @leszekswirski.

That means **30%** of the memory is taken by the VM for its internal execution support. That's a lot! The vast majority of these Code objects came from Full-Codegen and the IC (inline caching) system. V8 traditionally used to generate machine code for every function it executes, via the Full-Codegen compiler. That means even if the function is executed only once or twice during page load, we would still generate a Code object for it. And these code objects used to be really huge because Full-Codegen doesn't really apply any serious optimizations (it was supposed to generate code as quickly as possible). We added mitigations for this in the past, like a code aging mechanism, where the GC (garbage collector) would eventually nuke Code objects for functions that weren't executed for a certain period of time.

But even with these mitigations in place, the overhead of code generated for functions was significant. And the TurboFan optimizing compiler wouldn't help at all here. But as it turned out, some smart engineers figured out that we could reuse the actual code generation parts of the TurboFan pipeline to build the Ignition interpreter, which drastically reduces the code memory overhead. In addition to that it also improves page load time and helps to mitigate the parsing overhead, because the TurboFan optimizing compiler no longer needs to reparse the function source when it starts to optimize, but can optimize directly from the interpreters' bytecode.

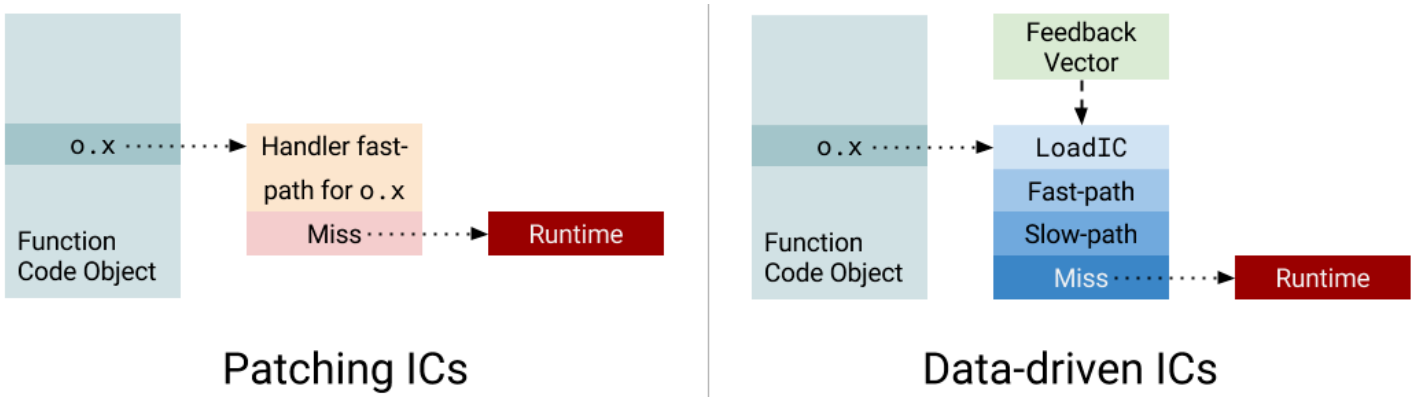


Source: V8: Hooking up the Ignition to the TurboFan, BlinkOn 7 conference, @rossmcilroy and @leszekswirski.

The interpreter is a big win for V8. But its impact on page load time and baseline performance is not overall positive. The issues with the slow paths, especially in the IC (inline caching) system, remain even with Ignition (and TurboFan). A key observation here, was that the traditional approach of having dedicated code stubs, so-called handlers, for the various combinations of maps (hidden classes) and names to speed up property accesses, doesn't scale. For example, for each property access `o.x` executed by V8, it would generate one Code object for each map of `o`, which checks whether `o` still has this map and if so, loads the value of `x` according to that map. The knowledge about the object and the way how to get to the property value was thus encoded in tiny Code objects. This contributed a lot to the general code memory overhead, and was also fairly expensive in terms of instruction cache utilization. But even worse, V8 would have to generate these Code objects for each and every property access that is executed at least two times (we mitigated overhead already by not doing this on the first execution).

Some web pages would spend a significant amount of time just generating these property access handlers during page load. Again, replacing the optimizing compiler wouldn't help at all, but instead we were able to generalize the TurboFan based code generation architecture that was introduced for Ignition to also be able to use it for code stubs. This

allowed us to refactor the IC system to move away from handler Code objects towards a data-driven approach, where the information how to load or store a property is encoded via a data format, and TurboFan based code stubs (i.e. `LoadIC`, `StoreIC`, etc.) read this format and perform the appropriate actions, utilizing a new data structure, the so-called FeedbackVector, that is now attached to every function and is responsible to record and manage all execution feedback, necessary to speed up JavaScript execution.



This greatly reduces the execution overhead during page load, and also significantly reduces the number of (tiny) Code objects. The new abstraction mechanism we build on top of the TurboFan code generation architecture is called the `CodeStubAssembler`, which provides a C++ based DSL (domain specific language) to generate machine code in a highly portable fashion. With this portable assembler, we can generate highly efficient code to even handle parts of the slow-path in JavaScript land without having to go to the C++ runtime (which is the really slow path).

There was a third area in V8, which traditionally suffered from unpredictable (baseline) performance: The builtins defined by the JavaScript language. These are library functions like `Object.create`, `Function.prototype.bind` or `String.prototype.charCodeAt`. Traditionally these were implemented in an awkward mix of self-hosted JavaScript, hand-written machine code (one for each of the nine supported architectures of V8), partial fast-paths in Crankshaft, and C++ runtime fallbacks. This was not only a serious source of correctness, stability and security bugs, but also one of the main contributors to generally unpredictable performance.

For example, using `Object.create` in a simple microbenchmark often gave pretty good performance, but as soon as it hit a real application, where you have a couple of different libraries using it and thereby feeding it with conflicting feedback, the performance dropped

significantly, and this feedback pollution led to performance drops in functions that would use the resulting objects. Nowadays `Object.create` is a so-called TurboFan builtin, based on the `CodeStubAssembler` technology, and provides predictable, decent performance more or less independent of the uses.

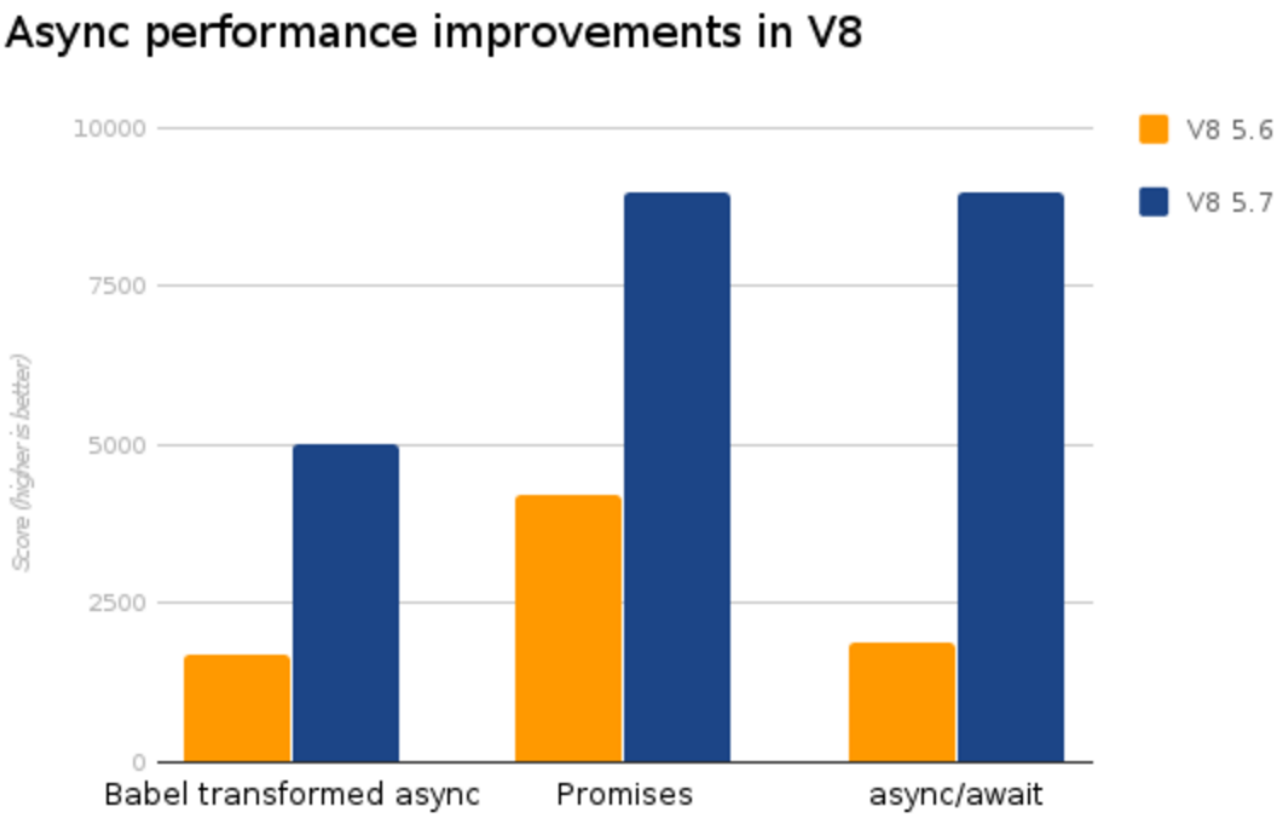
Another prime example is `Function.prototype.bind`, which was a pretty popular entry point to blaming V8 for bad builtin performance (i.e. John-David Dalton made it a habit to point to the poor performance of bound functions in V8… and he was right). The implementation of `Function.prototype.bind` in V8 two years ago was basically this:

```
// ES6 9.2.3.2 Function.prototype.bind(thisArg , ...args)
function FunctionBind(this_arg) { // Length is 1.
  if (!IS_CALLABLE(this)) throw MakeTypeError(kFunctionBind);

  var boundFunction = function () {
    // Poison .arguments and .caller, but is otherwise not det
    "use strict";
    // This function must not use any object literals (Object,
    // since the literals-array is being used to store the bou
    if (!IS_UNDEFINED(new.target)) {
      return %NewObjectFromBound(boundFunction);
    }
    var bindings = %BoundFunctionGetBindings(boundFunction);

    var argc = %_ArgumentsLength();
    if (argc == 0) {
      return %Apply(bindings[0], bindings[1], bindings, 2, bin
    }
    if (bindings.length === 2) {
      return %Apply(bindings[0], bindings[1], arguments, 0, ar
    }
    var bound_argc = bindings.length - 2;
    var argv = new InternalArray(bound_argc + argc);
    for (var i = 0; i < bound_argc; i++) {
      argv[i] = bindings[i + 2];
    }
    for (var j = 0; j < argc; j++) {
      argv[i++] = %_Arguments(j);
    }
    return %Apply(bindings[0], bindings[1], argv, 0, bound_arg
  };
```

```
  var proto = %_GetPrototype(this);  // in ES6 9.4.1.3 BoundFu

  var new_length = 0;
  if (ObjectGetOwnPropertyDescriptor(this, "length") !== UNDEF
    var old_length = this.length;
    if (IS_NUMBER(old_length)) {
      var argc = %_ArgumentsLength();
      if (argc > 0) argc--;  // Don't count the thisArg as par
      new_length = TO_INTEGER(old_length) - argc;
      if (new_length < 0) new_length = 0;
    }
  }

  // This runtime function finds any remaining arguments on th
  // so we don't pass the arguments object.
  var result = %FunctionBindArguments(boundFunction, this, thi
                                      new_length, proto);

  var name = this.name;
  var bound_name = IS_STRING(name) ? name : "";
  %DefineDataPropertyUnchecked(result, "name", "bound " + bour
                               DONT_ENUM | READ_ONLY);

  // We already have caller and arguments properties on functi
  // which are non-configurable. It therefore makes no sence t
  // try to redefine these as defined by the spec. The spec sa
  // that bind should make these throw a TypeError if get or s
  // is called and make them non-enumerable and non-configurab
  // To be consistent with our normal functions we leave this
  // TODO(lrn): Do set these to be thrower.
  return result;
}
```

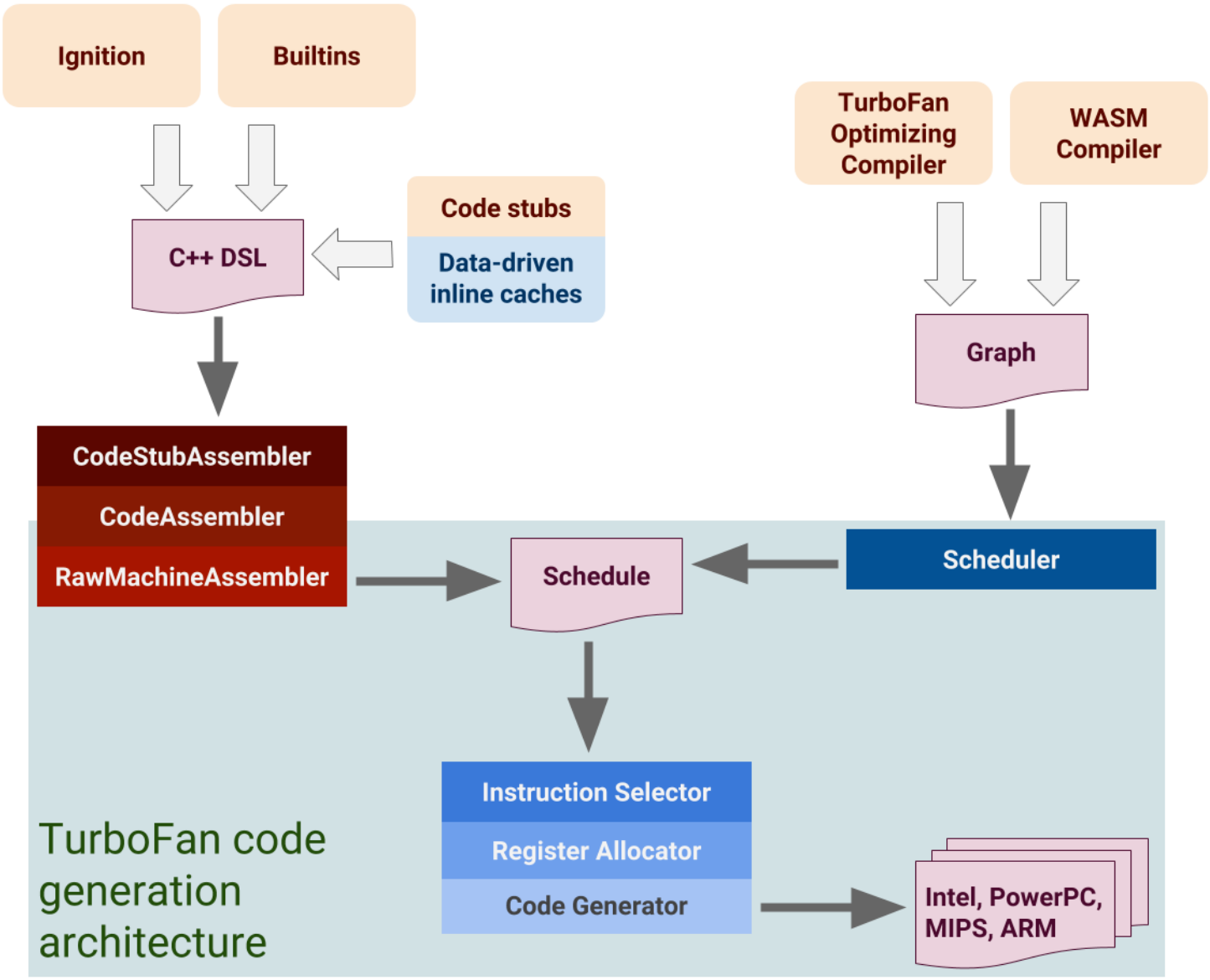
Note that `%Foo` is a special internal syntax and means call the function `Foo` in the C++ runtime, whereas `%_Bar` is a special internal syntax to inline some assembly identified by `Bar`. I leave it as an exercise to the reader to figure out why this code would be slow, given that crossing the boundary to the C++ runtime is pretty expensive (you can also read about it here). Just rewriting this builtin in a saner way (initially fully based on a single C++ implementation) and providing a simpler implementation for bound functions yielded **60,000%** improvements. The final performance boost was achieved when the `Function.prototype.bind` builtin itself was ported to the `CodeStubAssembler`.

Yet another example was the Promise implementation in V8, which was suffering a lot, and people would actually prefer to use polyfills despite V8 providing a native Promise implementation for quite some time. By porting the Promise implementation to the `CodeStubAssembler`, we were able to speed up Promises and async/await by **500%**.



Source: V8 release 5.7.

So despite being the most well-known component in the whole TurboFan story, the actual optimizing compiler is only one part of the puzzle and depending on how you look at it, it's not even the most important part. Below is a rough sketch of the current TurboFan code generation architecture. A lot of those parts are already shipping in Chrome today. For example a lot of the builtins have been utilizing TurboFan for quite a while, Ignition is enabled for low-end Android devices since Chrome 53, and most of the data-driven IC work is already available. Thus the final launch of the full pipeline is probably the most important event in the whole TurboFan story, but in some sense it's just the icing on the cake.



For me personally, this is just the beginning, as the new architecture opens a whole new world of possible optimizations for JavaScript. It will be exciting to push forward on optimizing the `Array` builtins, such as `Array.prototype.map`, `Array.prototype.forEach` and friends, and finally being able to inline them into TurboFan optimized code, which was more or less fundamentally impossible in Crankshaft for a couple of reasons. And I'm also looking forward to ways to further improve performance of new ES2015+ language features.

One thing that makes me very happy, is that onboarding new people on TurboFan technology was a much more pleasant experience than trying to onboard people on the weird mix of Crankshaft, Full-Codegen, self-hosted JavaScript, hand-written machine code and C++ runtime we had in the past.