JavaScript and V8 TurboFan

Aug 3, 2014 (/2014/08/javascript-and-v8-turbofan) · 5 min read · #javascript (/tags/javascript/) #performance (/tags/performance/) #v8 (/tags/v8/) #web (/tags/web/)

(/images/2014/08/raptor.jpg)Recently, Google engineers landed (https://codereview.chromium.org/426233002) a new optimizing JavaScript compiler for V8, codenamed **TurboFan**. As the name implies, this is supposed to further improve JavaScript execution speed, likely to be better than its predecessor, Crankshaft (http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html). While TurboFan is still in its early stage, that doesn't mean we can't take a look at it.

Playing with this TurboFan flavor of V8 is not difficult. First you need to build (https://developers.google.com/v8/build) the bleeding-edge branch (https://code.google.com/p/v8/source/browse/branches/bleeding_edge), where this 70,000-lines of code currently resides. After the new V8 shell is freshly baked, we can have some fun and inspecting TurboFan's work.

I did not have the time to dig really deep yet, so for now we just take a peek at the initial stage of the new optimizing compiler's pipeline.

Let's have a simple test program:

```
function answer() {
  return 42;
}
print(answer());
```

If this is `test.js`, then we can play with TurboFan by running:

```
/path/to/v8/shell --always-opt \
  --trace-turbo \
  --turbo-types \
  --turbo-filter=answer \
  test.js
```

We use the `--always-opt` option so that the code is optimized immediately (otherwise, only e.g. hot loops will be optimized). In order to inspect TurboFan, `--trace-turbo` and `--turbo-types` options are necessary. Last but not least, we are only interested in our own function *answer()* to be examined, hence the use of `--turbo-filter` option. If we pass * instead, V8 will dump too much information, mostly on other internals slightly irrelevant for this discussion.

We can see that TurboFan is doing its magic by looking at the first few lines of the output:

```
-----------------------------------------------
Begin compiling method answer using Turbofan
```

For further investigation, it is better to redirect the output to a log file. The log file will contain the data for 4 different graphs: **initial untyped** graph, **context specialized** graph, **lowered typed** graph, and **lowered generic** graph. This is the result of the Turbofan compiling pipeline. Every graph is easy to visualize since it is printed in the de-facto dot format (http://en.wikipedia.org/wiki/DOT_(graph_description_language)).

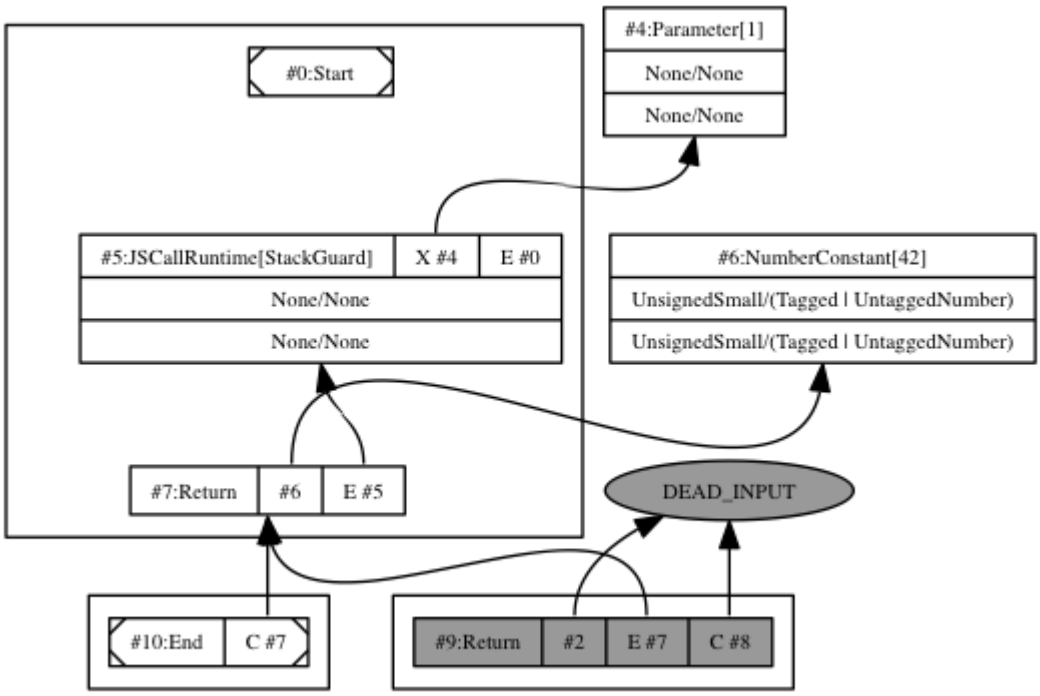
First, we need to separate each individual graph:

```
csplit -s -f graph log "/--/" {4}
```

Assuming GraphViz (http://en.wikipedia.org/wiki/Graphviz) is installed, we can see the first graph, the initial untyped, by running:

```
tail -n +2 graph01 | dot -T png > untyped.png
```

which is shown in the following screenshot:



(/images/2014/08/untyped.png)

This is the intermediate representation (IR) directed graph. You may recognize some nodes in the graph resembling the original JavaScript code such as the `NumberConstant[42]` and `Return` nodes. Each node has an operator and its associated IR opcode. This is very similar to the **Sea of Nodes IR** approach from Cliff Click (http://www.cliffc.org/blog/) (see Combining analyses, combining optimizations (http://dl.acm.org/citation.cfm?doid=201059.201061) and A Simple Graph-Based Intermediate Representation (http://www.oracle.com/technetwork/java/javase/tech/c2-ir95-150110.pdf)) used by Java HotSpot compiler (http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html).

The above graph is built as the first compiler pipeline stage by **traversing** the abstract syntax tree. There is hardly a surprise here. The edges *Start* (node #0) and *End* (node #10) are self explanatory. For every function, the information on its *Parameter* (node #4) is always mandatory. Checking the stack is an inherent part of V8 internals, hence the need for *JSCallRuntime* (node #5).

Inside the function body, every statement will be visited by the AST builder. In our example, there is only one, a return statement. For this, the builder also needs to visit the argument, which happens to be a numeric literal. The final outcome is a node which represents the opcode *Return* (node #7) which also refers to the constant (node #6).

The node in gray (*Return*, node #9) indicates that it is "dead", i.e. unused. This is actually a special return statement (returning undefined) which plays a role only if the function does not have an explicit return. Since it is not the case here, the node is not being used or referred anywhere, hence its dead status.

After this initial graph is obtained, the next stage are **context specialization**, **type analysis**, and **IR lowering**. These three topics are outside the scope (pun intended) of what I want to cover right now so we will have to discuss them some other time. However, note that our `test.js` is very simple, there is no assignment or any complicated operations and hence the subsequent compiler stages do not enhance the IR graph in any meaningful way. In fact, If you plot `graph02` (using the similar `dot` command as before), you will see that the resulting image is exactly the same as in the previous screenshot.

Ultimately, TurboFan needs to generate some machine code. Predictably, it has its own code generator (currently for x86 and ARM, both 32-bit and 64-bit), it does not reuse the existing Hydrogen and Lithium code generators from Crankshaft. The machine code is emitted from the instruction sequence. If you take a look at the log file, the relevant part of the sequence is:

```
14: ArchRet v6(=rax)
```

If you find out what *v6* is, it refers to the constant 42. On x86-64, this instruction thus can be turned into a `MOVQ RAX, 0x2A00000000` followed by `RET 8`. Straightforward, isn't it?

TurboFan is still very young and I'm sure there is still lots of room to grow. In most recent episode of JavaScript engine optimization, WebKit enjoys a speed boost thanks to the new FTL (https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/) (fourth-tier JIT compiler based on LLVM (http://blog.llvm.org/2014/07/ftl-webkits-llvm-based-jit.html)) while Firefox continues to refine (https://blog.mozilla.org/javascript/2014/07/15/ionmonkey-optimizing-away/) its whole-method JIT compiler IonMonkey (https://wiki.mozilla.org/IonMonkey). Will TurboFan become the V8's answer to them?

Welcome to the world, TurboFan!

Related posts:

Portrait Mode and JPEG Compression (/2018/04/portrait-mode-and-jpeg-compression)

Static Site with Hugo and Firebase (/2017/05/static-site-with-hugo-and-firebase)

Squeezing JPEG Images with Guetzli (/2017/03/squeezing-jpeg-images-with-guetzli)

Windows for Web Development (/2017/02/windows-for-web-development)

ChakraCore on Linux (/2017/01/chakracore-on-linux)

On-the-fly JavaScript Syntax Node Inspection (/2016/12/on-the-fly-javascript-syntax-node-inspection)

♡ this article? Explore more articles (/articles) and follow me Twitter (https://twitter.com/intent/follow?screen_name=AriyaHidayat).

%20V8%20TurboFan&url=https%3a%2f%2fariya.io%2f2014%2f08%2fjavascript-and-v8-turbofan&via=AriyaHidayat)

u=https%3a%2f%2fariya.io%2f2014%2f08%2fjavascript-and-v8-turbofan)

Subscribe via RSS (/index.xml)