Digging into the TurboFan JIT

Published 13 July 2015 · Tagged with internals

Last week we announced that we've turned on TurboFan for certain types of JavaScript. In this post we wanted to dig deeper into the design of TurboFan.

Performance has always been at the core of V8's strategy. TurboFan combines a cutting-edge intermediate representation with a multi-layered translation and optimization pipeline to generate better quality machine code than what was previously possible with the CrankShaft JIT. Optimizations in TurboFan are more numerous, more sophisticated, and more thoroughly applied than in CrankShaft, enabling fluid code motion, control flow optimizations, and precise numerical range analysis, all of which were more previously unattainable.

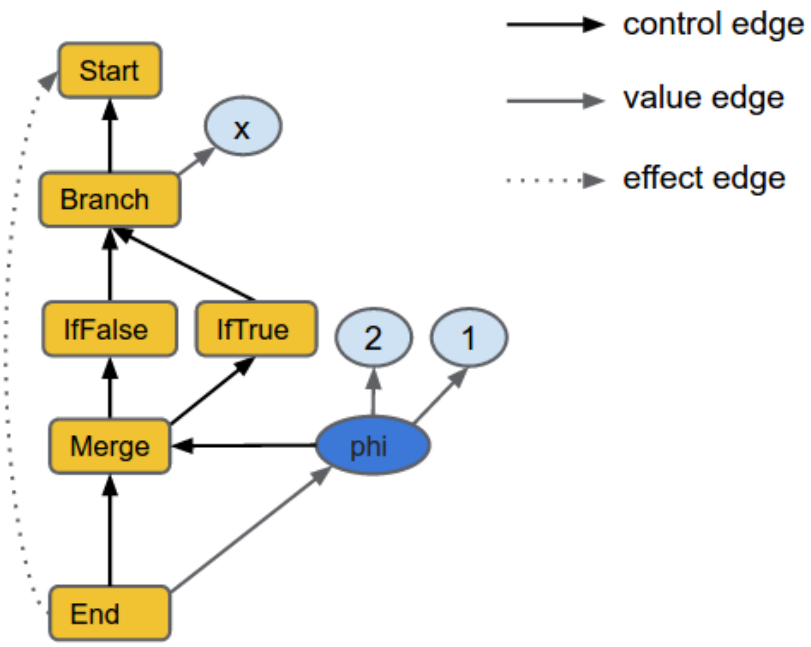
A layered architecture

Compilers tend to become complex over time as new language features are supported, new optimizations are added, and new computer architectures are targeted. With TurboFan, we've taken lessons from many compilers and developed a layered architecture to allow the compiler to cope with these demands over time. A clearer separation between the source-level language (JavaScript), the VM's capabilities (V8), and the architecture's intricacies (from x86 to ARM to MIPS) allows for cleaner and more robust code. Layering allows those working on the compiler to reason locally when implementing optimizations and features, as well as write more effective unit tests. It also saves code. Each of the 7 target architectures supported by TurboFan requires fewer than 3,000 lines of platform-specific code, versus 13,000-16,000 in CrankShaft. This enabled engineers at ARM, Intel, MIPS, and IBM to contribute to TurboFan in a much more effective way. TurboFan is able to more easily support all of the coming features of ES6 because its flexible design separates the JavaScript frontend from the architecture-dependent backends.

More sophisticated optimizations

The TurboFan JIT implements more aggressive optimizations than CrankShaft through a number of advanced techniques. JavaScript enters the compiler pipeline in a mostly unoptimized form and is translated and optimized to progressively lower forms until machine code is generated. The centerpiece of the design is a more relaxed sea-of-nodes internal representation (IR) of the code which allows more effective reordering and optimization.

function (x) { return x ? 1 : 2; }



Example TurboFan graph

Numerical range analysis helps TurboFan understand number-crunching code much better. The graph-based IR allows most optimizations to be expressed as simple local reductions which are easier to write and test independently. An optimization engine applies these local rules in a systematic and thorough way. Transitioning out of the graphical representation involves an innovative scheduling algorithm that makes use of the reordering freedom to move code out of loops and into less frequently executed paths. Finally, architecture-specific optimizations like complex instruction selection exploit features of each target platform for the best quality code.

Delivering a new level of performance

We're already seeing some great speedups with TurboFan, but there's still a ton of work to do. Stay tuned as we enable more optimizations and turn TurboFan on for more types of code!

Posted by Ben L. Titzer, Software Engineer and TurboFan Mechanic.