Sparkplug

Attention: Externally visible, non-confidential
Author: leszeks@chromium.org, verwaest@chromium.org
Status: Draft | Final
Created: 2021-02-02 / **Last updated:** 2021-02-17
Tracking Bug: https://crbug.com/v8/11420
Link: go/v8-sparkplug

LGTMs needed

Name	**Write (not) LGTM in this row**
hpayer	LGTM
rmcilroy	LGTM
mvstanton	LGTM
machenbach	LGTM
<your name here>	

TL;DR

Let's add a baseline compiler.

Overview

There is a trade-off space for compilers which balances compilation time and compiled code quality. The Ignition interpreter and the TurboFan optimising compiler are at two ends of this spectrum. However, there is a big performance cliff between the two; staying too long in the interpreter means we don't take advantage of optimisation, but calling TurboFan too early might mean we "waste" time optimising functions that aren't actually hot -- or worse, it means we deopt.

We can reduce this gap with a simple, fast, non-optimising compiler, that can quickly and cheaply tier-up from the interpreter by linearly walking the bytecode and spitting out machine code. We call this compiler Sparkplug.

The main properties of Sparkplug are:

- Fast to compile
  - No more than one or two passes over bytecode.
  - No intermediate representation -- machine code is dumped directly.
  - Fast enough compilation could mean we can flush Sparkplug code aggressively.
- Safe to compile early
  - No speculation, and therefore no deopts.
- Context independent
  - Therefore cacheable, both in-memory and on-disk.
  - Also shareable across navigations.
- Minimal complexity
  - Reuse as much existing machinery as possible (builtins, macro-assembler, stack frames).
  - Minimise architecture-specific code.
- Generates half decent code.
  - This is at the end, because while we do want to generate code that is efficient to run, we do not want to compromise on the above properties to do so.
  - Should “pay for itself” in CPU cycles, i.e. compilation (concurrent or not) should take less time than time saved by having better code.

Design

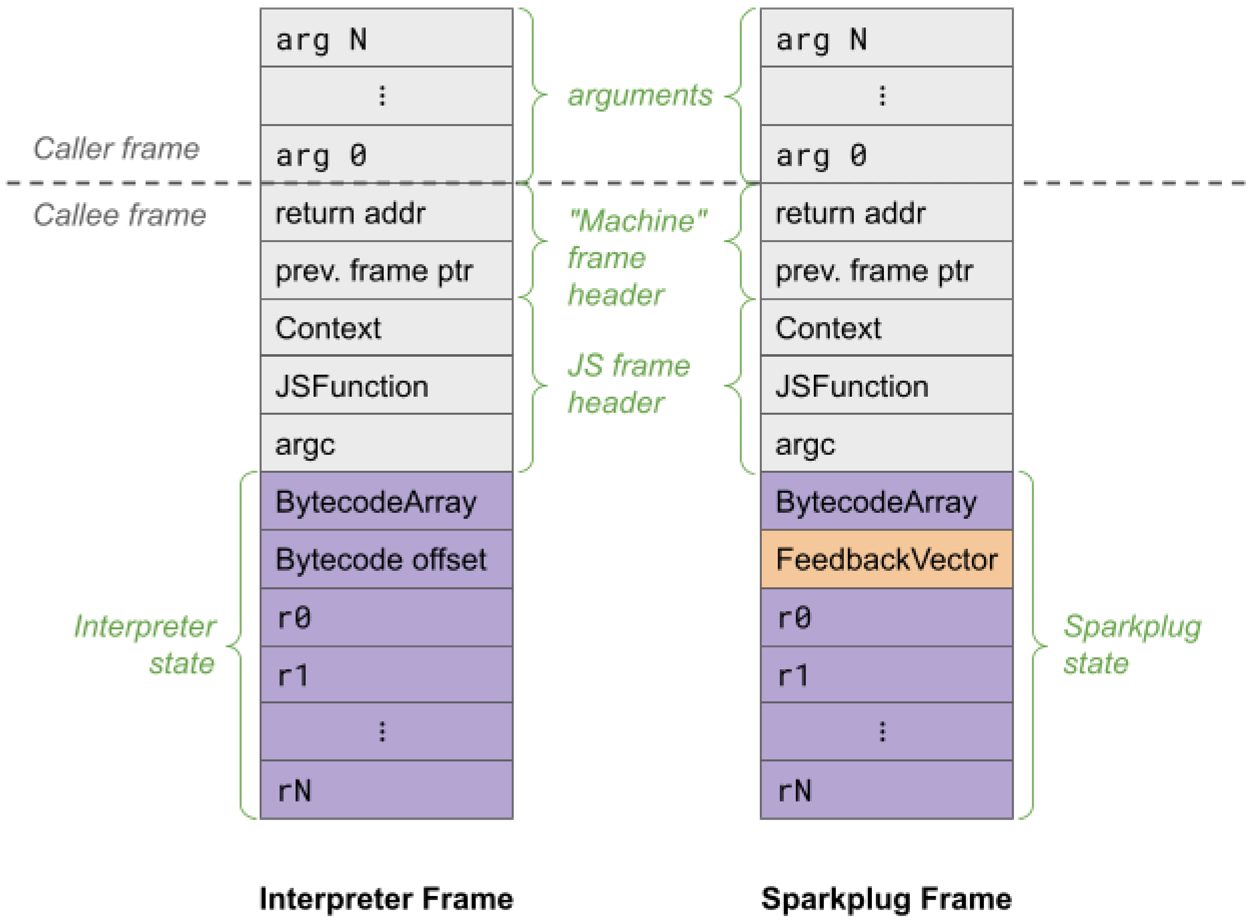
Overview

Sparkplug compiles bytecode to machine code. It can be thought of as an “interpreter accelerator”, which compiles bytecode in the most naïve way possible: into (effectively) a series of builtin calls interspersed with control flow.

More specifically, Sparkplug compiles from bytecode, by iterating the bytecode and emitting machine code for each bytecode as it is visited. Control flow is done in the obvious way; *very* simple operations, like reference equality or typeof tests, can be emitted directly; and more complex operations, like named loads or arithmetic, are punted to builtins.

Stack frames

Sparkplug avoids having to do any register allocation by re-using the interpreter's register frame, where bytecodes that load/store registers are compiled to direct loads/stores to the appropriate register frame stack slot. In fact, Sparkplug's stack frames are almost 1:1 compatible with Ignition stack frames:



The one difference is that it overwrites the bytecode offset slot (which is not needed) with a cached feedback vector.

This stack-level compatibility means that it is very cheap to OSR between the interpreter and Sparkplug, and the majority of frame inspection code (such as the stack frame printing or the GC's stack walk) can treat Sparkplug frames as interpreter frames. Similarly, Sparkplug can OSR up to the same TurboFan code as the interpreter does, using existing mechanisms, and TurboFan can deopt down to Sparkplug by synthesising almost exactly the same frame as it does for the interpreter.

Sparkplug's frame is initialised by a baseline prologue builtin, which sets up the fixed part of the frame, followed by an register frame fill which is inlined into the Sparkplug code. The inlining of the frame fill allows us to emit the exact number of pushes needed to fill the frame; if there are a lot of registers (>16) this is replaced by an unrolled loop pushing 8 registers at a time.

Sparkplug PC ↔ bytecode offset mapping

Since Sparkplug does not maintain a bytecode offset slot, we need another way of mapping sparkplug PCs to bytecode offsets (for frame walks, source positions, exceptions, OSR, etc.).

This is currently implemented as a "good enough" solution, which attaches a table to the Sparkplug Code object that encodes, linearly, pairs of diffs from the previous PC and bytecode offset. For example, the table

PC	Offset
20	0
30	3
50	5

is encoded as

PC	Offset
20	0
10	3
20	2

These diffs are then compressed with a variable-width integer encoding (specifically, VLQ) to minimise the table size. The pairs are recorded "wherever they are needed" -- practically, this means at loop headers (for OSR), immediately after calls (for exceptions), and at deopt points. Note that this diff-based encoding works because the Sparkplug PC increases monotonically with bytecode offset

This has the advantage of being simple, relatively compact, and two-way, but

- It requires a linear walk of the table to get the mapping,

- It “wastes space” on bytecode offsets which could instead be recovered from the BytecodeArray,
- We currently don’t get detailed line information when profiling Sparkplug

There are several potential improvements to this encoding, depending on the trade-offs we want to make:

- Performance: Encode the full PC and offset with a fixed size encoding, to make it binary searchable
- Performance: Add “checkpoints” to make it at least somewhat binary searchable
- Memory: Don’t encode bytecode offsets, but instead iterate the bytecode array while iterating PCs
- Memory: Generate this table lazily, similar to lazy bytecode source positions
- Memory: Have a PC “predictor” and encode the PC as an error from that prediction rather than a diff from the previous PC
- Memory: Compress the table and decompress on each use
- Profiling/Memory: Create full mapping only when profiling is enabled
- Profiling/Memory: Ensure at least one entry per basic block

Exceptions

Ignition implements exceptions as a handler table on the bytecode, mapping a bytecode offset range to the bytecode offset of the handler. TurboFan implements exceptions as a table from call return PC to handler PC.

Sparkplug takes a hybrid approach to the two, which uses the PC↔offset mapping above. When an exception is being processed, the stack walk finds the Sparkplug frame, and reads off the PC (which is necessarily a call return PC as Sparkplug can’t throw directly). It converts this PC to a bytecode offset, looks up that offset in the bytecode’s handler table, finds the handler offset, and converts that *back* into a Sparkplug PC, that it can then jump to.

This allows us to reuse the existing Ignition handler table with low additional memory cost (the cost of encoding the PC↔offset mapping). The remaining code (e.g. for handler lookup) stays the same as for interpreted frames.

Tier-up and interrupts

Sparkplug uses the same interrupt budget/interrupt mechanism as Ignition, Turboprop and NCI; that is, updating the interrupt budget by a fixed, bytecode-dependent amount on jumps and returns, and potentially calling a runtime interrupt on backward jumps and returns. Currently this interrupt budget update and check is inlined into the Sparkplug code; for code-size and compile-time reasons, we may move it to a (frameless) builtin call.

The tier-up from Ignition to Sparkplug is currently bolted onto lazy feedback vector allocation, and Sparkplug code is stored on the data field of the SharedFunctionInfo as a (Code, BytecodeArray) pair. This means that we do have to adapt some of the interpreter entries (in particular, the InterpreterEntryTrampoline) to check for Sparkplug code, and tail-call it if it exists. This is folded relatively simply into the existing BytecodeArray load from the SharedFunctionInfo.

Note: for Sparkplug we changed the lazy feedback vector allocation heuristics to be in relation to the size of the function rather than a constant budget. The cost of compilation is in relation to the function size for both, and this modification helps both reduce the number of IC misses and allows Sparkplug to benefit earlier and more.

Feedback vectors

Sparkplug requires the function's feedback vector to be allocated, to be able to remove feedback cell checks, or indeed indirect access to the feedback vector through the feedback cell. This is ensured at tier-up time, both as part of Sparkplug compilation, and in the Interpreter→Sparkplug tier-up in the InterpreterEntryTrampoline.

Sparkplug maintains the same feedback as Ignition, so most builtins called from Sparkplug require a feedback vector (ICs, Call trampolines, arithmetic, etc.). We currently re-use the "_WithFeedback" builtins originally created for NCI, however this requires us to emit code which loads the feedback vector inline in the Sparkplug code. The plan is to move Sparkplug to call specific "Baseline" trampoline builtins, which load the feedback vector from the stack themselves. This is already done for "hot" builtins, like LoadIC, Call, and Construct.

Loops / OSR

Sparkplug is currently a two-pass compiler; the reason for the first pass is loops. Specifically, a first walk over the bytecode is needed to discover JumpLoop bytecodes, and thus find loop headers. If we wanted, we could skip this pass by emitting a Label for *every* bytecode. However, we may want to have a two-pass compiler for other reasons too, e.g. liveness estimates.

OSR in loops is performed exactly the same as in the interpreter; it checks whether OSR is armed on the bytecode, and tail-calls the OSR trampoline if yes.

Debugger

The plan for the debugger is, like TurboProp/TurboFan, to discard the optimized code and fall back to the interpreter. This means a relatively trivial OSR fixup of any on-stack Sparkplug frames.

Cross-arch code

The Sparkplug compiler uses a “BaselineAssembler” which wraps the MacroAssembler. This BaselineAssembler provides an architecture-independent interface for performing common Sparkplug operations, like calling builtins, loading/storing registers, comparing values, doing jumps, and so on. The majority of the implementation of the BaselineAssembler can also be written in a (syntactically) cross-arch method, relying on the different MacroAssembler implementations having the same syntactic API.

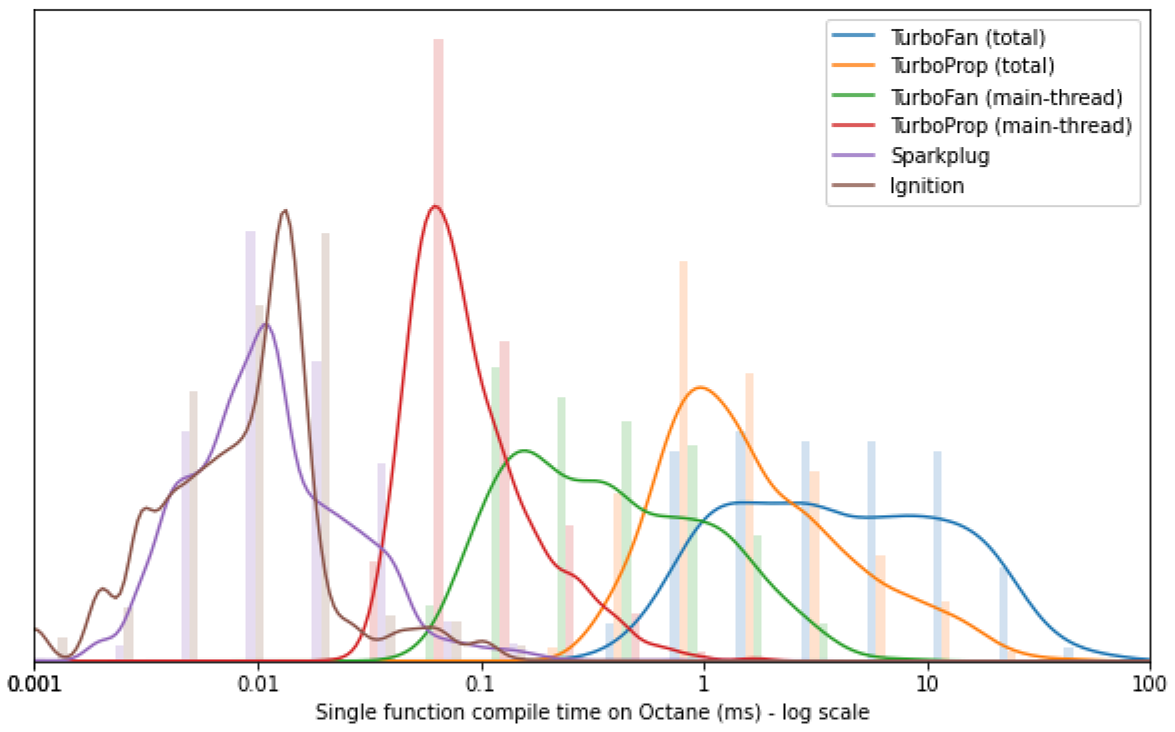
Code that _has_ to be architecture specific, like pushing arguments for a call (thanks, Arm64 stack alignment), or the actual operation of loading a value at an offset from the stack frame, lives in architecture-specific -inl.h files. The aim is to keep these below 1k LoC.

Performance

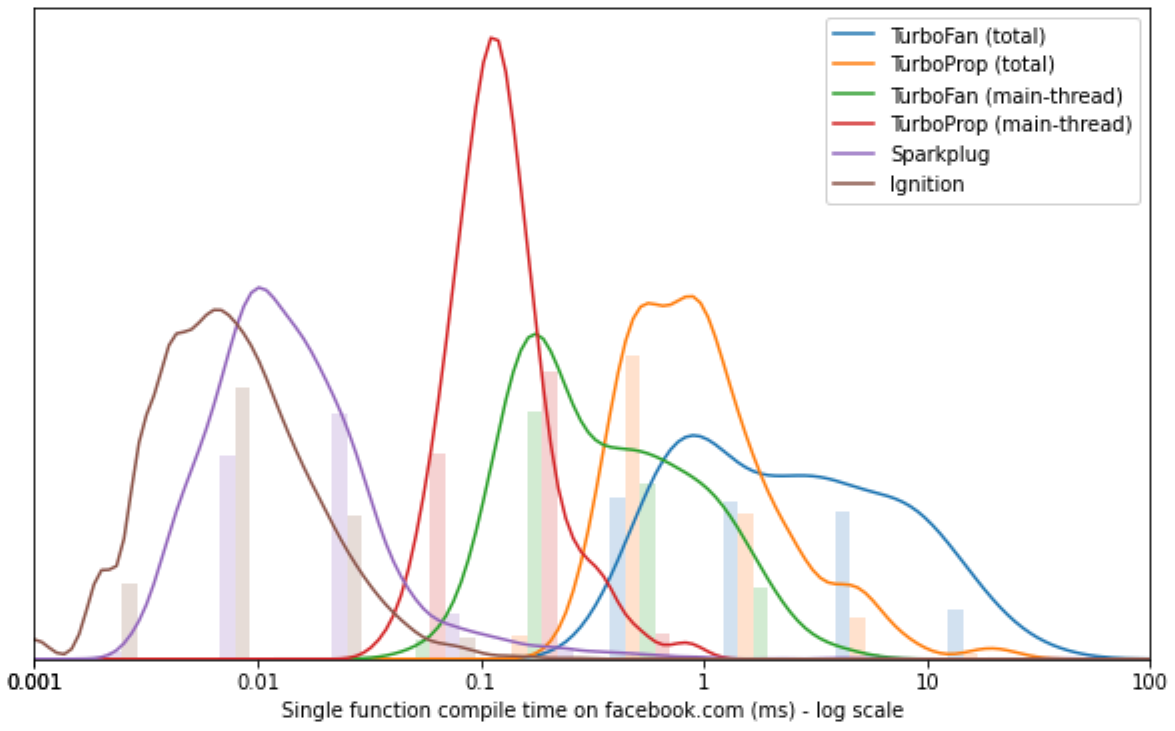
We have an initial prototype, which can be used to gather some preliminary data. Our current prototype has the following performance characteristics:

Compile time

Based on tracing compilation time in Octane, on a powerful workstation compile time is, roughly, 100-1000× faster than TurboFan/TurboProp’s *total* execution time (including background-thread), and, roughly, 10-100× faster than their (current) main-thread time (note: this was measured without “concurrent inlining” or “direct heap access”). As a comparison, Sparkplug compilation time is on a similar order of magnitude to bytecode compilation (which excludes parsing time).



On Facebook we see a similar distribution, albeit with bytecode compilation being slightly faster than Sparkplug compilation:

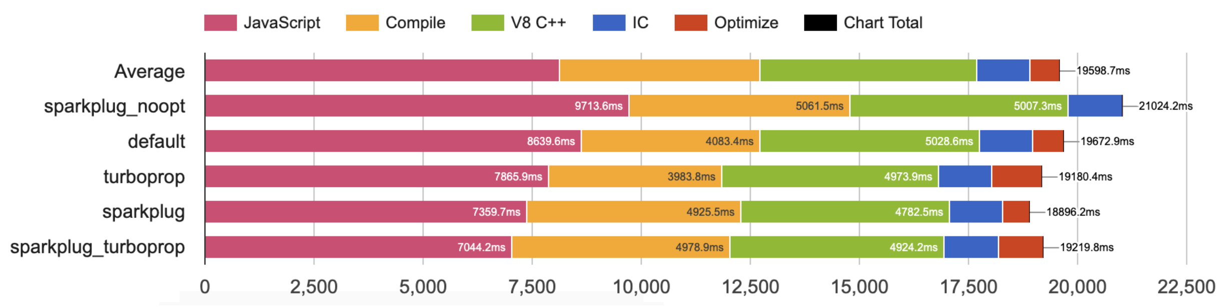


Note: These numbers were collected with a PGO-enabled build of Chromium, but without training PGO on Sparkplug. This likely biases the numbers against Sparkplug compilation time.

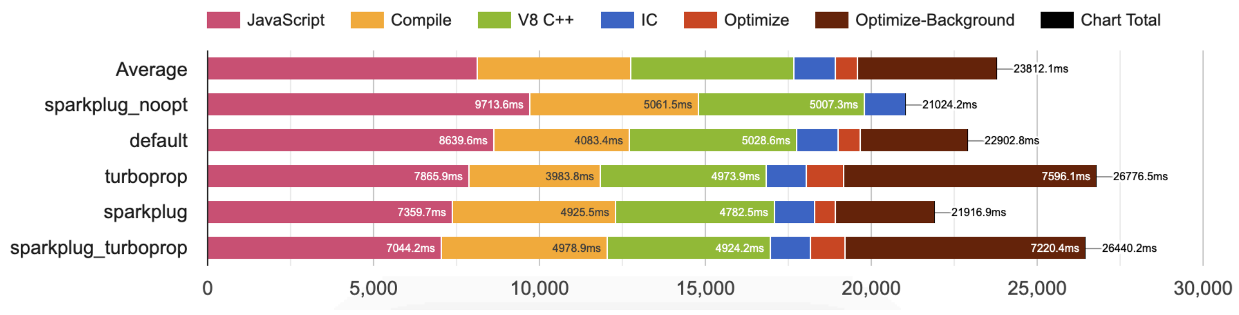
Real-world (system_health RuntimeCallStats)

We ran Sparkplug on the system_health loading stories, with RCS, to see its effect on both compile time and javascript execution time. For comparison, we also ran the current default configuration, TurboProp, and a combination of both Sparkplug and TurboProp.

Running on a Mac, 6×2 CPUs, with 3× repetitions, the relevant parts of main-thread time look like this:



If we include time spent optimizing on the background, we see this:



Note: Both Interpreter and Sparkplug compile time are bucketed under "Compile", Turboprop compile time is bucketed under "Optimize" for the main-thread part, and "Optimize-Background" for the background thread part.

Notable here is that:

- Sparkplug already "pays for itself", in terms of the sum of compile time and JavaScript execution time. This is expected to improve as we improve both its compile time and the quality of generated code.

- Sparkplug doesn't increase total compile time *that much*, on top of existing interpreter compile time; indeed, looking at compile counts, the default configuration has 35,971 compiles while Sparkplug has 42,020, indicating that with the current heuristic, ~20% of functions are compiled.
- Sparkplug by itself (without even TurboFan) has the lowest total CPU usage, which is valuable on pay-per-cycle servers and battery-powered devices.
- Turboprop pairs well with Sparkplug to give the lowest JS execution time

Real-world memory (system_health)

Sparkplug code is approximately 5-6× larger than the corresponding bytecode, but with the current tier-up heuristic we compile ~20% of functions, so overall it regresses ~2× relative to bytecode memory (excluding flushing). Measurements on Pinpoint don't report statistically significant increases in V8 heap memory or malloc memory; presumably we'd see regressions with stronger statistical testing, but it seems low enough to not be immediately obvious.

Security

Sparkplug generates machine code, but isn't especially creative about it; most of the code it generates is stack access and calls, and the generation code is short enough to manually verify. So, there are unlikely to be security issues in the generated code itself.

Interaction with the rest of the system is the larger risk. Having a similar (but not quite the same) layout as interpreted frames means that potentially we could get type confusion on the mismatches (either the feedback vector or the bytecode offset). This is somewhat unavoidable (since we cannot maintain the invariant that frames are valid in both), so we rely on DCHECKs around the frame type and fuzzing to flush out issues here.

The other concern is W^X. Flipping the write/execution bits is a small percentage of TurboFan compile time, but dominates Sparkplug compile time due to Sparkplug compiling faster (flipping bits roughly doubles Sparkplug compilation time on Speedometer). A major source for performance issues on at least Intel CPUs seems to originate from micro-architectural SMC (self-modifying code) snooping optimisations (fast path from the dcache to the icache) that only kick in when the written-to code memory is actually executable. Flipping between RX (execute) and RWX (write) instead of RW seems to recover ~70% of the overhead. While this might be enough to recover lost performance on higher-end machines, on lower-end we likely need a performance-friendlier way to implement W^X (likely some sort of concurrency and/or batching) as we may want to age code aggressively to reduce memory overhead.

Testing plan

Sparkplug is currently hidden behind two flags: `--sparkplug` and `--always-sparkplug`. The former adds Sparkplug with the tiering described above; the latter forces Sparkplug compilation immediately after bytecode compilation, to increase coverage.

The testing plan is to add two variants to the test runner, enabling each of these flags. These variants would start as fyi/extra initially, but the `--sparkplug` variant should be added to the default set of variants run on waterfall/CQ blocking bots. One plausible option is to put `--sparkplug` behind `--future`.

When `--sparkplug` becomes the default, we add a variant with `--no-sparkplug` or cross-combine it with an existing variant.

We should also add the sparkplug to the fuzzer, both for “normal” fuzzing and for correctness fuzzing (comparing outputs against the interpreter). We enable normal fuzzing with trials for both flags above on Clusterfuzz and for V8’s flag fuzzer. For differential fuzzing we’ll add specific comparison configurations.

To test on fuzzers before landing, we can upload a custom debug binary on Clusterfuzz to the linux_d8_dbg_cm job.

FAQ

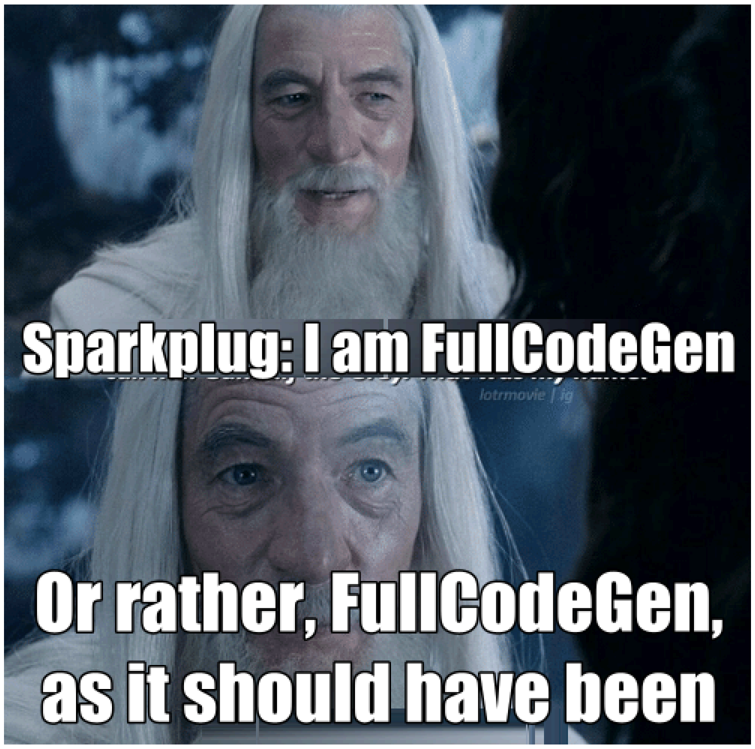
Isn’t this just FullCodeGen?

Kind of! It’s FullCodeGen for modern V8, and that’s not such a bad thing. This brings the performance benefits of FCG back to V8, but

- Compiles from bytecode, so there’s no need for another AST walk, desugaring, etc.
- Keeps bytecode as the “source of truth”, so no need to e.g. maintain consistent deopt IDs between bytecode and compiled code. Basically, no “Frankenpipeline”.
- Uses data-driven ICs from the get-go, so no complexity around code patching.
- Defers most complex work to builtins, which can be written cross-arch in Torque/CSA.

- Maintains the interpreter’s stack layout (see Stack frames) so OSR stays easy.

In other words:


Sparkplug: I am FullCodeGen
lotrmovie | ig
Or rather, FullCodeGen, as it should have been

Isn’t this just TurboProp?

TurboProp is also designed as a mid-tier compiler between the interpreter and TurboFan, but it approaches this middle from the TurboFan side, as a “TurboFan-lite” that re-uses the existing TurboFan machinery, but removes less necessary optimisation passes, and replaces slower-but-more-optimal passes with faster-but-less-optimal ones.

TurboProp is faster than TurboFan, but it is still a “somewhat optimising” compiler, which has to set up the Sea-of-Nodes IR, schedule the graph, allocate registers etc. As argued in The Case For Four Tiers (Google internal), this makes it a great choice for a pre-TurboFan optimisation tier, but there is still room between Ignition and Turboprop for a non-optimising compiler, which approaches the mid-tier from the Interpreter side.

Furthermore, the presence of Sparkplug can “take the weight” off Turboprop; if Turboprop isn’t the first tier after Ignition, it has a little bit more flexibility in exchanging compile time for performance (e.g. inlining) or being a little bit more selective in what functions it compiles.

Are Sparkplug frames considered “interpreted”?

“Jein”[1]. At the moment, they mostly are (since the have mostly the same layout, this simplifies compatibility with the rest of the system which doesn’t really care). We probably want to extract out a concept of an “unoptimized” JS frame, which covers both interpreted and sparkplug frames.

Is Sparkplug code considered “optimized”?

“Jein”[2]. It’s definitely more “optimized” than the bytecode, has tier-up from bytecode and even tier-down to bytecode when the debugger sets a breakpoint in the code. On the other hand, when we talk about “optimized” we normally really mean “speculating”, in the sense that e.g. changes in object shapes could cause the code to deopt. In this sense it is “unoptimized”.

We’ll likely want to do a similar thing to frames; expand the definition of “unoptimized” to include both interpreted and sparkplug, but distinguish sparkplug from interpreted where it matters (e.g. tier-up). Likely this will involve sparkplug being its own thing, IsBaseline or similar.

[1] My favourite German word, meaning “yes and no”.
[2] Still my favourite German word.