Status: Draft | Review | Work-in-progress | Final
Author(s): seth.brenith@microsoft.com
Last update: 2020-04-10
Tracking bug: v8:10470

Profile-guided optimization of V8 builtins

Attention: Shared externally

Name	**LGTM** or **NOT LGTM w/ reason**
sartang@microsoft.com	LGTM
jgruber@chromium.org	LGTM
mvstanton@chromium.org	LGTM
<add yourself>	

Summary

I propose adding an option to collect basic block usage data from V8 builtins which can be fed back into a subsequent run of mksnapshot, where it affects the ordering of basic blocks within the builtins. This procedure is completely optional and does not affect any V8 builds that don't opt in with a gn argument. At a high level, the steps match what one might expect based on other PGO tools:

1. Build V8 with instrumentation enabled
2. Run whatever scenarios you think are representative of your workload; V8 emits files with the collected profiling data
3. Build again, including the files from step 2 as an input

Design

What data is collected

In this proposal, the profiling data is a 32-bit counter for each basic block in each builtin, representing how many times that basic block was run.

How data is collected

The existing BasicBlockInstrumenter, which can be requested with --turbo-profiling, does almost what we need: for every basic block in TurboFan-generated code, it adds a few extra instructions to load a counter from memory, increment it, and store it back. However, it accesses that data by using raw pointers into C++ structs allocated on the system heap, which is not a safe strategy for code written during mksnapshot and later deserialized from the snapshot. After deserialization in a new process, those C++ structs don't exist, and even if we attempted to recreate them, there's no guarantee that we could allocate them at the same addresses. Instead, the instrumentation logic should write to somewhere in the JS heap, so that the pointers can get relocated properly. I see a couple of options here, and there may be more:

1. **One big array of counters for builtins.** Since builtins are compiled serially, we can easily assign a range of indices to each builtin, based on the number of basic blocks in that function. Once all of the builtins are created, we know how many counters are required in total, so we can allocate a ByteArray big enough to hold all of the counters. In the future, if mksnapshot is updated to compile builtins in parallel, we could disable parallel compilation for PGO builds to maintain this simplicity.
2. **Reimplement existing BasicBlockProfiler::Data as a JS heap object.** Currently, BasicBlockInstrumenter stores a lot more than just counters: it includes the function name, a string representation of the schedule, and some other metadata that helps V8 to emit useful messages correlating the counters to the schedule. Ideally, all of this extra data could be stored in the JS heap instead, so that instrumentation of builtins and runtime-generated code can behave identically. This would require allocating a bunch of things on the JS heap during scheduling, and I'm not familiar with how to do that correctly, but I believe it's possible. This is the option I prefer.

After data is collected in-memory, it needs to be written to a file during process shutdown. Isolate::DumpAndResetStats seems to be a reasonable place for this. The normal v8.log file can be used as the output, provided that we include recognizable messages for the beginning and end of the profiling data (since other data may also be included in the log).

How data is used

Because snapshot construction is deterministic, the same basic blocks get created in the same order and assigned the same IDs upon a subsequent run, which allows us to easily match them up with the profiling data. Only one change is made based on the profiling data: some basic blocks are marked as deferred during CFGBuilder::ConnectBranch, if they are substantially less likely than the other branch outcome. With this change, the likely path through a function becomes closer to straight-line code, which is best for instruction cache usage and static branch prediction. (The current prototype defines a branch outcome X as substantially more likely than the alternate outcome Y if X/4 > Y and X > 100.) This implementation has the benefit of avoiding any extra complexity, since TurboFan already knows about scheduling blocks that have been marked as deferred.

Does it work?

I tested a prototype implementation by profiling on JetStream 2 and then comparing JetStream 2 scores on a build with PGO builtins to an unmodified build. In this case (the optimal case, where usage exactly matches the training), the overall score improved by 1.6%. Tests from the Web Tooling Benchmark fared particularly well: Acorn, Chai, CoffeeScript, Lebab, TypeScript, and Uglify all saw 3-5% improvements. It's not immediately apparent what level of improvement this strategy could yield for general web browsing, but this feature would give embedders the freedom to optimize V8's builtins in whatever way is most appropriate for their usage.

Future ideas

We could imagine various similar optimizations, which aren't covered in this proposal. Thanks to Tobias for providing these ideas.

- Count branch outcomes in the interpreter and feed that information into scheduling for compiled JavaScript functions
- Use low-rate sampling of real-world usage as the data source (there would be some difficulty in correlating data from one version of V8 with the code in a new version, where both the builtins and the compiler could have changed)

- Emit warnings if a branch hint provided in CSA mismatches the profiling data
- Use profiling data to recommend places where macros should become builtins or vice versa (since all inlining decisions in Torque/CSA are made by the programmer rather than the compiler)