Maglev

Attention: Externally visible, non-confidential

Author: jgruber@chromium.org, leszeks@chromium.org, verwaest@chromium.org
Status: Draft | Final
Created: 2022-02-04
Tracking Bug: v8:7700
Link: go/v8-maglev

LGTMs needed

Name	Write (not) LGTM in this row
hpayer	LGTM
tebbi	LGTM
saelo	LGTM
<your name here>	

Note: This is a long doc and incomplete on the design details. LGTMs are for the summary and milestones.

TL;DR

Let's add a mid-tier optimising compiler designed mainly for compilation speed that can still generate good code for straightforward JS.

Summary

Overview

We've previously made a case why four tiers in V8(Google internal) make sense to explain why Sparkplug made sense in addition to Ignition, TurboProp and TurboFan. TurboProp was a midtier compiler proposal based on TurboFan to significantly improve compilation speed while compromising on the performance of the resulting code. With Sparkplug in place, however, the design tradeoffs TurboProp made by being built on top of TurboFan didn't end up panning out. While the resulting compilation speed was a massive improvement over TurboFan, Sparkplug

showed there was still a ~100x compilation speed gap between Sparkplug and TurboProp even though that configuration of TurboProp did not inline functions at all. However, we still observed that having TurboProp in the 4-tier configuration still improved performance, and would improve performance further if it compiled faster.

Instead of finding a slimmer subset of TurboFan to use as a mid-tier, we propose to build a minimal SSA-based optimising compiler from the ground up tuned for compilation speed. The target is not more than 5-10x slower than Sparkplug.

The main properties of Maglev are:

- Fast to optimise / reoptimise after deoptimisation
- SSA macro-instructions in a CFG, with no distinction between high- and low-level
- Phases are single-pass, forward graph walks where possible
- Minimal number of passes over the IR
- Efficient diff-based deopt/checkpoint encoding
- Data-driven ICs to improve front-end performance
- Designed for concurrency

Alternative Considered: Sparkplug with feedback

An alternative design we considered is adding IC feedback and speculation to Sparkplug; that is, create a compiler which lowers each bytecode directly to machine code and maintains the interpreter register frame, but additionally make the machine code lowering perform speculative optimisations and allow it to deopt back down to the interpreter (or to non-speculating Sparkplug).

This sort of compiler could do the obvious speculating optimisations (e.g. map check + direct field load), and basic forward-propagation optimisations (like map check elimination, some basic form of GVN). It would, however, struggle to do more advanced optimisations that require code motion or backwards passes, like hoisting or representation analysis. We therefore decided that it would be unlikely to provide a sufficient code-gen improvement over Sparkplug to be worth an extra tier – we may revisit this decision in the future, however, either as a tier-3 Maglev alternative if we feel that Maglev compile costs are too high, or as a tier-2 Sparkplug enhancement.

Milestones and Tasks

Upstreaming (Mid-to-late Feb 2022)

… the first version pushed to the main V8 repository. The code layout should be relatively stable (to avoid churn on the main repo), and it compiles and runs basic example code + simple tests on CQ. Code shall be pushed to the main V8 repository behind a gn/runtime flag. x64-only.

- Basic implementation / code layout
- Simple tests which can run on the CQ

“Parallelisable work” MVP (Late Feb/Early March 2022)

… a base for further development. APIs and infrastructure are sufficiently complete s.t. development can be increasingly parallelized and more developers can be pulled in. Still x64-only.

- Basic APIs and implementation available for
    - the IR
    - the graph processor
    - the register allocator (together with requirement specification on IR)
    - code dependencies

Full x64 prototype (2022Q2? 2022Q3?)

... a fleshed out version of the MVP. Runs multiple useful benchmarks line items in a realistic setting. All Maglev components exist at least with a basic implementation.

- A real register allocator
- Inlining
- Graph building optimisations, e.g. map check elimination
- Concurrent compilation
- Representation inference
- Code lifecycle / tiering
- (Fast) deopts
- Shared deferred code?
- Efficient safepoint generation and encoding (delta-encoding?)
- OSR?
- Fuzzing and perf infrastructure

Architecture ports (EOY 2022?)

… the x64 prototype ported to other architectures. Work on the non-codegen parts of the prototype can continue, but we push to make the backend work on the other supported architectures.

Ready for production (2022? 2023?)

… a production quality implementation. Finching and the shipping process will likely start at this milestone.

- Optimised heuristics (tierup, inlining, …)
- All critical optimization passes implemented

- Thorough and robust benchmarking result

Design

Detailed design decision follow – these are subject to change as we go, so consider them a mostly up-to-date summary of the current state rather than decisions set in stone

Overview

Maglev has a single SSA (static single assignment) CFG (control flow graph) IR (intermediate representation), with no separation of “high” or “low” level nodes. In principle, any node generated during graph building can be emitted during code generation.

Basic blocks are split into

- a “header” - currently just a list of Phi values entering this block
- a “body” - a list of the value/effect nodes (no control flow)
- a “control node” - a conditional or unconditional jump node with one or more targets.

Basic blocks are topologically sorted in control-flow order, so that all non-loop jumps are forward jumps, and all non-loop phis refer to values in previous blocks. This sorting comes naturally out of graph building, and the initial basic block layout order matches that of the bytecode; later phases are expected to preserve this order.

Nodes are relatively large, with space for various annotations (e.g. register allocation), and it’s preferred to mutate/annotate nodes rather than replace them. They optionally provide a value, and optionally have an effect. Nodes are strongly typed in C++, and we have a (mostly flat) class hierarchy with a base Node class, and a subclass for each opcode.

Nodes have an opcode, an input count, and store a fixed-size set of input pointers (whose size is fixed on Node construct time). Inputs are stored in memory in reverse order *before* the node – this allows walking inputs in generic code without having to read and switch on the opcode (only having to read the input count).

Phis have special handling compared to other values. Rather than being normal value nodes, they are saved as a linked list in the basic block’s header. Each Phi has an input per predecessor feeding into it – basic blocks that jump to a merge block (i.e. to a basic block with multiple predecessors) store a “predecessor id” which corresponds to the index of the value this block provides in the target’s phis’ input lists. Note that this means that only unconditional jumps can provide phis, i.e. that only unconditional jumps can have merge targets. This is enforced by splitting critical edges (edges where the predecessor has multiple successors, and the successor has multiple predecessors) by inserting empty basic blocks.

Alternative Considered: Separate high-level and low-level IR

- Slimmer high-level nodes, by using side tables for the various node annotations, or “pointer compressing” node pointers.
- Requires copying and backlinking; more space for “surviving nodes”
- May make sense to separate out low-level details if we want to reuse our IR as the high-level IR for a faster fourth tier compiler. It might also not matter if we simply ignore the additional data, and/or copy to a different IR.

Alternative Considered: Different basic block design

- TurboShaft style node management
    - May not be necessary for the first IR in the compiler; we don’t yet know how many nodes we’ll have; we can copy out anyway

Graph Building

The Maglev graph is built in a similar way to the Turbofan graph builder, as an abstract interpretation of the bytecode with appropriate merging of jump targets.

Nodes

As we walk bytecode, we create a new SSA node for each value producing bytecode. For operations that store/load registers, we maintain an InterpreterFrameState (IFS) which, for each register (and the accumulator), stores a pointer to the Node currently stored in that register.

On control flow, the IFS is copied and stored in the “merge target” (keyed by bytecode offset), to be picked up once that target bytecode is reached. When multiple jumps target the same bytecode (or control flow falls through into a merge target), the IFS are “merged”, which means creating a Phi for each register in the IFS. We create Phis ”on demand”, i.e. only when the register’s Node is different between the two IFS, to avoid creating Phis for values that stay unchanged across the control flow.

Loop Phis are created on loop header IFS allocation; we know where loop headers are from a bytecode analysis prepass (the same bytecode analysis as in TurboFan). This prepass also collects per-bytecode liveness information, and there’s a prepass that counts how many merge targets each bytecode offset has, so that we can pre-size Phis.

Basic blocks

Nodes are emitted into basic blocks, which are built as we go. We start a new basic block at each merge target (incl. loop headers) and end it at jumps (either because of a Jump* bytecode, or because a merge target is hit and we start a new basic block). Merge targets keep a list of predecessor basic blocks, and blocks that jump to a merge target are given a “predecessor id” which represents their index in each Phi’s input list.

Note that this means that blocks jumping to merge targets can only have one successor (in other words, that there can be no critical edges). We ensure this by inserting empty blocks whenever the predecessor has conditional control flow.

Checkpoints / Interpreter state

~~Interpreter state is encoded in the graph in two ways. First, whenever there is a store to an interpreter register, we emit a StoreToFrame node which represents a virtual store to a register in the interpreter frame. Secondly, whenever there is a potentially deopting operation, we emit a Checkpoint node, which stores the current accumulator value and the current bytecode offset.~~

~~Thus, subsequent passes over the IR can reconstruct the interpreter state (registers, accumulator and bytecode offset) at deopt points by creating a new IFS and replaying these stores. We avoid having to replay IFS splitting/merging by storing a snapshot of the IFS at each merge target basic block (i.e. each basic block that isn't a simple fallthrough).~~

Nodes optionally contain an eager or lazy deopt, which stores a snapshot copy of the full interpreter frame at the time the node was emitted (eager deopts can be deduplicated with earlier ones if there were no side effects between). Previously we considered a design where this storage was incremental and reconstructed in later phases, but this proved to be less efficient as we ended up eventually creating the flattened deopt anyway.

Immediate lowering

Whenever we visit a bytecode with feedback, we immediately process that feedback and try to lower that bytecode to as specific an operation as possible. This is currently the only use of feedback in the compiler, and is likely to stay that way, with subsequent passes relying on static analysis rather than feedback data.

The plan here is to use as much IC information as possible, e.g. using the data cached in handlers to decide what nodes to generate, rather than re-calculating it from the Maps in the feedback. This will hopefully be slightly faster, since the ICs already did all the hard work.

Code dependencies are handled the same as in TurboFan, reusing the heap broker's dependency mechanism.

Optimisation phases

Yup, we'll definitely have some of these. It's not yet decided what they'll be though, the current thinking is some forms of representation analysis, GVN and potentially a minimal amount of loop hoisting.

Node processing

The plan is for the majority of these operations to be single forward passes over the graph. To assist with this, we have a node processing API which allows one to write a per-opcode Node processor/visitor that encapsulates the per-Node logic, and then pass that to a graph processor which walks the graph, switches on the current Node's opcode, and dispatches to the Node processor. The graph processor can also handle maintaining the implicit bytecode frame state for checkpoints, and the transitions between different basic blocks.

Additionally, there's a Node multi-processor which applies a set of Node processors in sequence to a Node. This can then be passed to the graph processor as a single Node processor, to allow performing multiple processing operations in one forward pass.

The graph processor and node multi-processor are both templated on the node processors, so hopefully the C++ compiler will be able to inline, hoist, and deduplicate the node processing code where it can.

Code Generation

Machine code is generated in three final phases: preprocessing, register allocation, and finally actual machine code generation. These are all done in a single pass using the above Node processing API.

Preprocessing

Preprocessing is a single linear pass which does three things: assigns monotonically increasing IDs to nodes, calculates live ranges of nodes, and resolves input/output requirements.

Assigning IDs

Assignment of monotonically increasing IDs is self explanatory – all nodes, whether effect, value or control, receive an ID. We use these IDs later for specifying and comparing nodes' live ranges. We have to do this now instead of during graph building in case there was any hoisting previously.

Calculating live ranges

Calculating live ranges is done ignoring control flow. Each node is assumed to start its live range at its location in the CFG, and end it at the last use of that node – the start and end are the IDs created above. This means that we can calculate live ranges simply by walking forwards and updating the "last use" ID each time it is used.

The special cases here are Phis and deopts. Phis set their inputs' last use to the predecessor block's control flow node, with the dual intention of a) restricting the input's lifetime to the point where it is assigned to the Phi (i.e. the end of the predecessor basic block), and b) ensuring that

inputs into loop Phis have their last use at the end of the loop (after their definition), not at the start of the loop.

Deopting operations don't store a direct list of Node pointers representing the interpreter frame state, but instead are considered to use the current implicit frame state as collected via basic block snapshots, StoreToFrame nodes, and Checkpoint nodes. Therefore, some care has to be taken to also extend these lifetimes as appropriate.

Input/output requirement resolution

This phase is tightly coupled with code generation, and is a pre-register-allocation resolution of what the requirements of the codegen are on the inputs and outputs – for example, whether a Node takes an input via an arbitrary register or via a specific one, or whether it outputs its value also into an arbitrary register. This phase fills in these requirements, which the register allocator can then read and resolve to specific registers.

Register allocation

There is a custom, simple register allocator, which performs a linear scan over the Nodes. Each Node has a canonical spill slot location from which it can be loaded, and its value may be cached in a register. The spilling to the spill slot currently happens at Node creation, and can be elided if the Node never needs to be spilled.

The current machine frame state is propagated forwards through the graph (similar to the interpreter frame state in graph building), and used to discover what Nodes are currently cached in which registers. On basic block merge points, the incoming machine frames are merged (where values mismatch) by adding a "register merge" to the block.

This approach should be equivalent to "standard" linear scan implemented with a separate mapping of use intervals, live ranges, etc., but instead of creating a parallel data structure representing the graph, it reuses the existing Node SSA IR and adds an abstract machine interpretation frame for propagating information forwards.

An additional capability is that Nodes can request a certain count of temporary registers as a separate concept to inputs. The register allocator ensures that there are enough free registers to fulfil this request, spilling as necessary.

Machine code generation

Machine code is generated in a final single pass, directly generating code with the MacroAssembler, with a relatively fixed "template" for each Node (this is similar to Sparkplug). The register allocator has already provided all the registers, so any register manipulation inside this code template should make sure to restore previous register state.

The code generation is also implemented using the graph processor described above, and therefore gets its interpreter frame simulation for free. This simulation is used for emitting deopt points, which materialise the interpreter frame from the current state.

This phase is expected to generate any deferred code needed for its operation – this is more similar to what Crankshaft used to do, and unlike TurboFan, which has an explicit “deferred” flag in its IR. Deferred code generators are pushed to a queue, and emitted after the graph has been visited – there is a JumpToDeferredIf helper for creating these deferred code generators from a non-capturing lambda, with some magic to help pass in arguments, deep copying where needed (in particular, deep copying interpreter frame states if the deferred code needs them for a deopt).

Deoptimisation and StackMaps

Incomplete…

- Checkpoint management
- diff encoding deopt data
- raw & dead values in the stack

Try/Catch/Finally

Incomplete…

- CPS & lazy deopt info
- Initially just try/finally + deopt

Representation Inference

Incomplete…

- We presume we need it
- We don’t have it yet

Inlining

Incomplete…

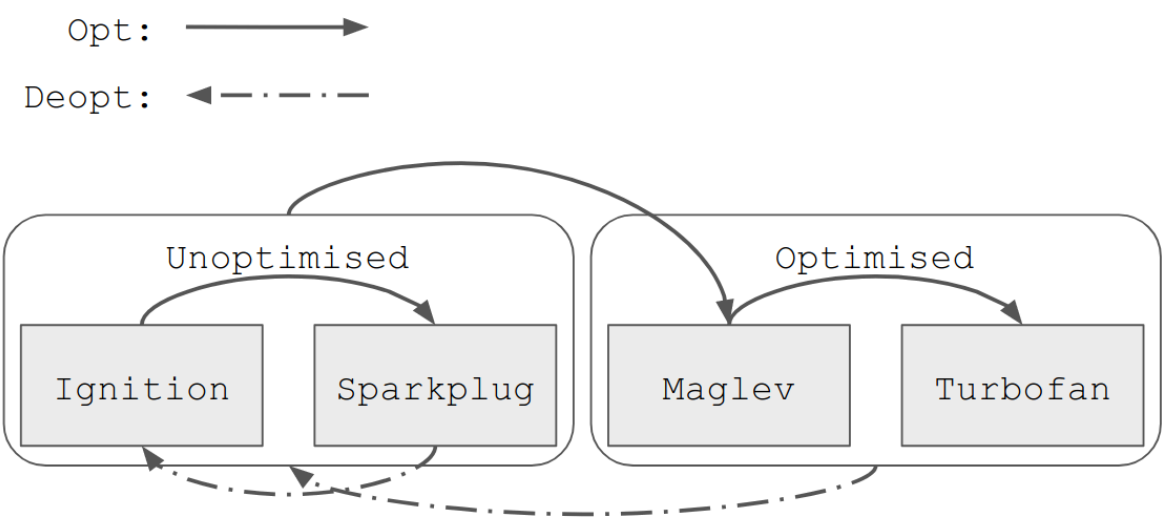
- Presume we need it.
- A place to be very careful since it easily explodes compilation times.

Compilation Dependencies

Incomplete…

- Two potential purposes: 1. guard assumptions during concurrent compilation, 2. install deopting dependencies after compilation.
- Reuse TF's CompilationDependencies class (potentially with specialized subclasses for TF/ML deps).
- Be careful not to create / install too many (TF has this issue).
- Use a generic create-commit workflow from the start (s.t. it's possible to abort a specific optimization and release related deps without installing them).

Pipeline Design



With Maglev there are a total of four tiers, split into unoptimised (Ignition and SP) and optimised (ML, TF) categories. The two unoptimised tiers roughly share frame layout and thus jump between Ignition and Sparkplug almost at will (OSR from Ignition into a plain/non-specialised SP Code object, deopt from SP into Ignition).

Unoptimised tiers tier up Maglev, Maglev tiers up to Turbofan. Both Maglev and Turbofan deoptimise into the highest available unoptimised tier.

OSR may eventually be enabled for all optimisation edges. Initially, the Maglev prototype only supports OSR from unoptimised tiers into Turbofan.

Debugging a function means tiering down to Ignition, in which we're able to set breakpoints, etc.

Heuristics are TBD but should be somewhat simplified / unified vs. the current implementation. Ideally, tierup heuristics between all tiers should use a similar mechanism, with a scale factor applied.

Relation to TurboShaft

Incomplete…

Performance

Compile time

Currently the early prototype for simple functions compiles in around 4x Sparkplug. While not complete, since we do already have quite a bit of the infrastructure it seems plausible that we'll manage to stay within 5-10x Sparkplug for simple functions; assuming we're thoughtful about inlining.

Testing, Fuzzing, Performance Tracking

As a new compiler tier, Maglev will require thorough stability and performance testing.

Dedicated Maglev tests are in `test/mjsunit/maglev/`. These are used mostly as development aids (e.g.: can we compile a loop?) and may or may not live beyond later milestones.

The primary way to test Maglev will be to **run our existing V8 test suites with dedicated Maglev configurations**:

- --maglev --disable-turbofan. In this configuration the pipeline is Ignition-Sparkplug-Maglev, i.e. all code that currently exercises TF will exercise ML.
- Just --maglev. The pipeline is Ignition-Sparkplug-Maglev-Turbofan. This is the intended final pipeline. In addition to the above, this also exercises tierups from ML to TF.
- The above, plus --always-maglev. To increase coverage, this could e.g. spawn ML compile jobs whenever bytecode compilation completes. We miss out on feedback, but gain more coverage of the compiler (this is useful since we know mjsunit tests have very incomplete coverage of optimized code).
- The above, plus --no-concurrent-recompilation to test both concurrent and non-concurrent optimization.

Out of the 8 possible configurations above, we should identify 2-3 critical ones to enable on bots. I would suggest 1) a `always_maglev` configuration as –maglev –always-maglev –disable-turbofan, and 2) a `maglev` configuration which initially runs –maglev –disable-turbofan but will later switch over to just –maglev.

Maglev can be enabled through a runtime flag --maglev (initially only on x64). **New Maglev testing variants should be added to selected x64 waterfall and CQ bots**.

Sanitizer bots (TSAN, MSAN) and fuzzers should be extended to run the new Maglev flag configurations. Especially TSAN is important for reducing bug-load for concurrent compilation.

For **Chromeperf**, it *may* be enough to piggy-back on top of --future, depending what else is currently enabled there. If the overhead is reasonable, it would be much preferable to have a dedicated --maglev mode to avoid interference from other in-progress work.

Finching: yes, once we get closer to completion.

UMA counters: yes, once we get closer to completion. Compile counts, times, …

UMA profiler: Maybe? Can it easily be taught to distinguish between ticks SP/ML/TF code?