Code generation for Isolate field accesses in WebAssembly-Turboshaft

Attention - this doc is public and shared with the world!

Contact: ahaas@
Contributors: <name3>, <name4>, …
Status: Inception | Draft | Accepted | Done

LGTMs needed

Name	**Write (not) LGTM in this row**
<LGTM provider 1>	
<LGTM provider 2>	
...	

Objective

The direct goal I want to solve is to pass a pointer to the isolate from generated code to a C++ function in a fast API call. In assembly code that is easy, take the root register, add an offset, and the result is a pointer to the isolate. However, telling Turboshaft to generate this code for WebAssembly is not straightforward. Any access to the isolate is cumbersome to generate at the moment for WebAssembly. With this design doc I want to start a discussion on whether we should refactor how accesses of isolate fields are generated for WebAssembly, or if we should just extend the current approach to allow loading a pointer to the isolate as well.

Background

The Isolate is the core data structure in V8, and is accessed a lot from generated code, both from JavaScript and WebAssembly. There is a special register, called the root register, that is used to access fields on the Isolate. Note that for efficiency, the root register does not point to the beginning of the isolate but to an offset in the middle. Thereby some fields in the isolate get accessed with negative offsets, and accesses to fields with small negative offsets is faster than with larger positive offsets.

When a JavaScript compiler wants to access a field on the Isolate, it creates an `ExternalReference` for that field, which is essentially just a wrapper around the address of that field in a particular instance of the isolate. In the macro assembler, when this address should be loaded into a register, a load relative to the root register is emitted if the offset of the address from the particular Isolate instance is small enough.
Note that even though the Isolate is used in this process twice, first to acquire the address of the field, and later to check the offset of an address from the isolate, the isolate itself actually never gets dereferenced.
In WebAssembly, compilation happens completely in the background, and the compiler does not have access to an Isolate. To emit a load relative to the root register in Turboshaft for WebAssembly, the `LoadRootRegisterOp` operation is used with the offset of the field from the root register. Currently `LoadRootRegister` is only allowed as the base for load or store operation.
This approach for WebAssembly has several downsides:

- For any Isolate field that gets accessed from WebAssembly code, the offset has to be calculated explicitly. In other words, for each field some boilerplate code is needed that is not needed for JavaScript.
- The current approach works well for loading from a field in the Isolate, but it does not work for loading a pointer to the Isolate itself into a register.

Example for boilerplate code:

```
None
class StackGuard {
...
  static constexpr int jslimit_offset() {
    return offsetof(StackGuard, thread_local_) +
           offsetof(ThreadLocal, jslimit_);
  }
...
};

class IsolateData {
...
  static constexpr int jslimit_offset() {
    // `stack_guard_offset()` already gets defined in a macro.
    return stack_guard_offset() + StackGuard::jslimit_offset();
  }
...
};

// In Turboshaft:
__ Load(
   __ LoadRootRegister(), LoadOp::Kind::RawAligned().NotLoadEliminable(),
   MemoryRepresentation::UintPtr(), IsolateData::jslimit_offset());
```

Design

Here are some ideas that allow us to reach the objectives mentioned above, with different trade-offs:

Continue using the current system

We could just continue to generate code that accesses isolate fields the same as we did so far, and just add code that allows us to load a pointer to the isolate into a register. During graph construction, we would probably add code as follows:

```
C/C++
__ WordPtrSub(__ LoadRootRegister(), Isolate::isolate_root_bias());
```

So far `__ LoadRootRegister()` was only allowed to be used as the base of a memory load or a memory store. The instruction selector marked `VisitLoadRootRegister()` as `UNREACHABLE()`. It would therefore be necessary to add platform-specific support in the instruction selector and code generator to support `LoadRootRegister()` in binops as well. I prototyped the changes, on x64 the changes were quite clean, on arm64 I ran into problems and started looking for alternatives 😅.

Pro:

- No changes needed in Turboshaft, only in the instruction selector and code generator

Con:

- The changes in the instruction selector and code generator may be more difficult than with root-offset constants.
- Requires platform-specific changes.

Root-offset constant

At the moment there exist `ExternalConstants` in Turboshaft for `ExternalReferences`. In addition to `ExternalConstants` we could introduce `RootOffsetConstants`. In the compilation phases these constants would be handled the same as `ExternalConstants`, but in the instruction selector and code generator these constants would be emitted as `root_register + offset`.

Pro:

- A pointer to the isolate can be loaded into a register this way.

- JavaScript and WebAssembly would be handled quite similarly, both would use a `Constant` in the IR, even though JavaScript uses an `ExternalConstant` and WebAssembly uses a `RootOffsetConstant`.

Con:

- An extension to the `ConstantOp` operation in Turboshaft is needed, additionally platform-specific changes in the instruction selector and code generator.
- JavaScript and WebAssembly would still be handled differently.
- Boilerplate code to calculate the offsets of fields on the isolate would still be needed for any fields that get accessed from JavaScript.
- Requires platform-specific changes.

~~Fake Isolate~~

~~As mentioned above, when code gets generated for JavaScript that accesses the Isolate, the pointer to the Isolate is used but never dereferenced. For WebAssembly we could therefore use a fake isolate that is just used for address calculations, but that is not allowed to be accessed. For example, we could use an address in the middle of some inaccessible memory. If the Isolate gets accessed, a segfault would be triggered. If the Isolate pointer is just used to calculate addresses and offset, everything would just work. Note that even `nullptr` would probably work.~~

~~The pointer to the isolate could be called something like `fake_isolate` or `inaccessible_isolate` to indicate clearly that this pointer cannot be accessed.~~

~~Pro:~~

- ~~`ExternalReferences` and accesses to fields on the isolate could be generated in WebAssembly as easily as in JavaScript.~~
- ~~The amount of changes needed for this approach would be minimal. The Isolate already exists in the compilation pipeline, it would just need to be initialized with a fake address instead of `nullptr`.~~

~~Con:~~

- ~~This approach is a hack. It may be surprising to see for developers that a pointer is used even though it is invalid.~~

[Update]: I prototyped this approach, and it works on x64 with some small changes, but I don't think it works on arm. The problem is, how can the compiler decide which constants should be loaded relative to the root register, and which constants should be loaded as absolute values. On x64 there is a heuristic which says that every `ExternalReference` within an `int32_t` offset from the isolate root should be loaded relative to the root register. On x64 this heuristic may be okay, it may be possible to place the isolate at a location in the address space that is not within 2GB of any isolate-independent object. However, on arm32 the heuristic is the same, and with the limited address space it is not possible to place the isolate far away from isolate-independent objects.

Note that for JS, an incorrect heuristic is not a problem, as the code is only used for the one isolate that it got compiled for. For WebAssembly, however code may be used with multiple

isolates, so it's important to load only those addresses relative to the root register that are isolate-dependent.

Extend the system with IsolateAddressId

The isolate has a field called `isolate_addresses_` that stores the addresses of other isolate fields. Additionally there is the enum `IsolateAddressId` that provides named indices into `isolate_addresses_`. It is therefore possible to store an `IsolateAddressId` in a constant in the Turboshaft graph, and later translate it to an actual address somewhere in the code generator.
The problem with `IsolateAddressId` is that you need an isolate to translate the id into an actual address, and then further into an offset from the root register. However, there could also be a direct translation from `IsolateAddressId` to a root offset, without the intermediate step of looking up an actual address in the middle. I have to admit though, in its current implementation I wonder why `IsolateAddressId` exists in the first place.