Status: Draft | Review | Work-in-progress | Final
Author(s): paolosev@microsoft.com
Last update: 2021-02-22
Tracking bug: none yet

Inlining Wasm functions

Name	**LGTM** or **NOT LGTM w/ reason**
duongn	LGTM
neis@chromium.org	
ahaas@chromium.org	
mvstanton@chromium.org	
…	

Summary

Is it possible to minimize the overhead of calling Wasm code from JavaScript by inlining (small) Wasm functions?

Overview

Ideally, calling into WebAssembly from JavaScript should be very fast, but currently this is not always the case in V8.

For example, let's say we have a WebAssembly function to multiply two integer numbers, like this:

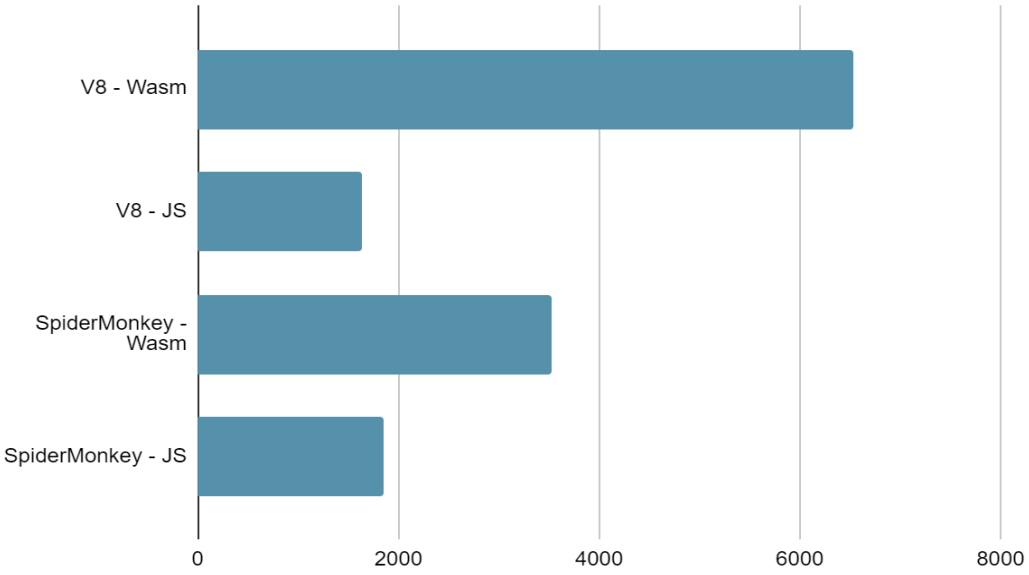
```
int square(int x, int y) {
  return x * y;
}
```

and say we call that from JavaScript in a tight loop like this:

```
...
let instance = new WebAssembly.Instance(module, importObj);
let wasm_square = instance.exports.square;

function js_square(value) { return value * value; }
```

```
function test() {
    let result = 0;
    for (let i = 0; i < 1e9; i++) {
        result = wasm_square(i % 999);

        // JS alternative:
        // result = js_square(i % 999);
    }
    return result;
}

let start = Date.now();
test();
let end = Date.now();
print(" time: " + (end - start) + " msec.");
```

We see that the Wasm version is ~4x slower than the pure JavaScript implementation (where we call a JavaScript version of function “square” instead of a Wasm function):

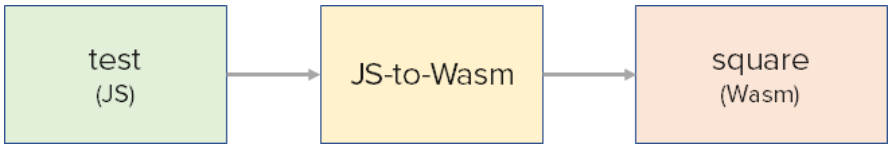


We also see that V8 is slower than SpiderMonkey, where some work already went into this optimization, as explained in the blog post Calls between JavaScript and WebAssembly are finally fast 🎉.

While this is clearly an extreme example, it shows that there is an overhead in Wasm calls that should be minimized, especially when the call is made from TurboFan optimized code.

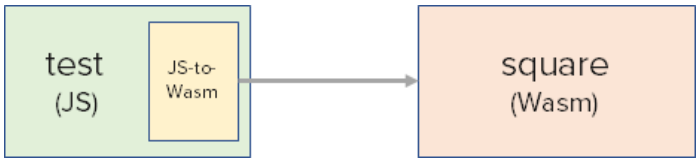
There are two reasons for this overhead:

1. TurboFan does not optimize calls to a Wasm function, but instead treats these calls as function calls to a JS`Function`, the so-called *JS-to-WASM wrapper*, which:
    a. Unboxes all parameters, converting them from tagged types into WebAssembly native types (i32/i64/f32/f64).
    b. Sets the “thread_in_wasm” flag in TLS, used for trap handling
    c. Calls the Wasm function.
    d. Resets the “thread_in_wasm” flag in TLS, used for trap handling
    e. Boxes the result into a tagged type.

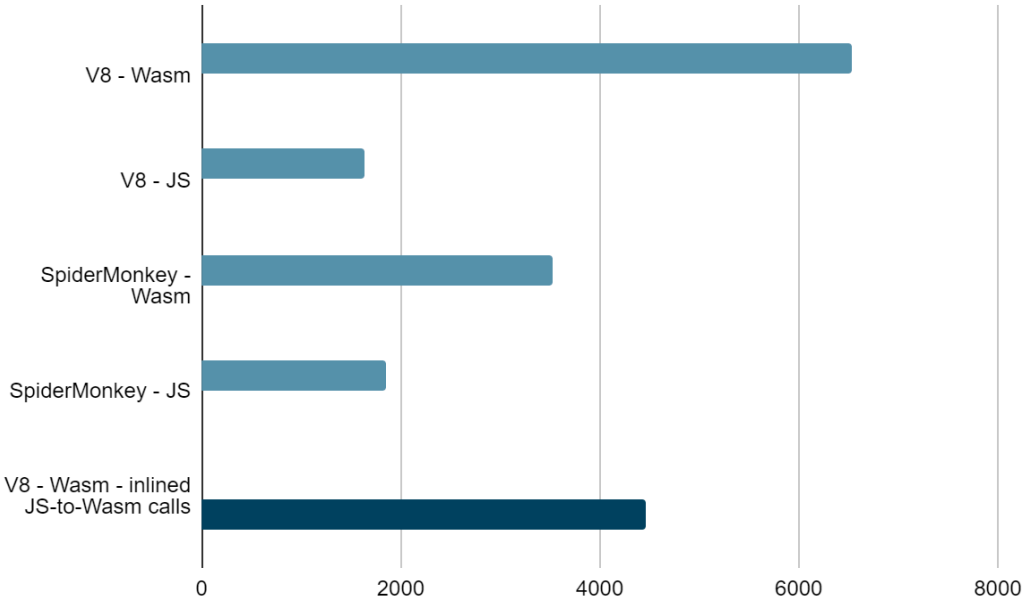


2. TurboFan can easily inline small JavaScript functions, like the function `js_square` in the previous example, so avoiding all overhead caused by a function call.

We fixed the first issue by inlining the JS-to-Wasm wrapper at the call site (see CL2596784 and CL 2599266). More details can be found in the Faster JS to Wasm calls doc.

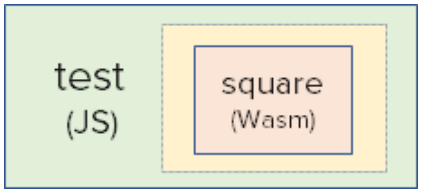


This allowed us to make Wasm calls quite faster:



:

But there is still a performance gap comparing JS->Wasm calls with pure JS->JS calls that we would like to address. Can we act on the second reason for the slowdown and **inline small Wasm functions** in TurboFan optimized code?



Implementation

The implementation of a small prototype for Wasm function inlining can be found in this CL 2609912.

This is built upon a version of V8 already modified to support the inlining of JS-to-Wasm wrappers (described above). Inlining not only the wrapper but also the Wasm function itself is the logical extension of that idea.

For this, we introduce two new operators:

- `JSWasmCall`, which represents a call to a JS-to-Wasm wrapper (made from a JavaScript function)
- `WasmCall`, which represents a call to a Wasm function (made from a JS-to-Wasm wrapper).

The initial idea in CL2596784 for the inlining of JS-to-Wasm wrappers was to implement inlining as part of simplified-lowering. This works but is unnecessarily complicated; the biggest problem is that we need to replicate in simplified-lowering the inlining logic already implemented in `JSInliner` and `JSInliningHeuristic`, with the additional complication that classes `SimplifiedLowering` and `RepresentationSelector` do not inherit from `Reducer` or `AdvancedReducer`, so they don't provide all the functionalities to edit a graph that are expected by the `JSInliner` code. We can do better by reusing the existing inlining logic provided by `JSInliner`.

For this reason we introduce a new TurboFan phase, named **wasm-inlining** that runs just after simplified-lowering and is logically similar to the [JavaScript] **inlining** phase we run at the beginning of the pipeline but works on a "lower-level" IR, where the "WasmMachineCode" graph generated by the Wasm compiler can be inlined in the graph of the caller function.

Implementation details

1. In the TurboFan **js-call-reducer** phase, we detect when a `JSCall` is directed to a JS-to-Wasm trampoline. The `JSCall` node is converted into a JSWasmCall. The associated parameter type, `JSWasmCallParameters`, contains all the data required to inline the Wasm call:
    - A pointer to the Wasm `NativeModule`
    - The index and signature of the Wasm function to call.
2. In the **simplified-lowering** phase, we lower the `JSWasmCall`, appropriately setting the `UseInfos` that correspond to the type each argument of the Wasm function and to the result type.
3. When WebAssembly inlining is enabled, we run the new **wasm-inlining** phase of the TurboFan pipeline. The phase contains a subset of the reducers we have in the **inlining** phase: only `DeadCodeElimination`, `CommonOperatorReducer` and `JSInliningHeuristic` (which here only considers `JSWasmCall` and `WasmCall` nodes).
   In `JSInliningHeuristic`:

    - For a **`JSWasmCall`** node, we call `WasmWrapperGraphBuilder::BuildJSToWasmWrapper` to generate the IR for the JS-to-Wasm wrapper, and inlines it (with `JSInliner::InlineCall`).

      This is all we need to do if we just want to inline the wrapper.

      If we also want to inline the called function, we transform its `Call` node into a `WasmCall` node, which will be then visited as part of the same graph traversal.

- For a **`WasmCall`** node, we call `BuildGraphForWasmFunction` to generate the graph for the Wasm function.

  A `WasmCall` is associated with a parameter type `WasmCallParameters` which is like `CallParameters` but contains additional data required for this compilation like the Wasm function index and signature and a pointer to the Wasm `NativeModule`, from which we get the function bytecode.

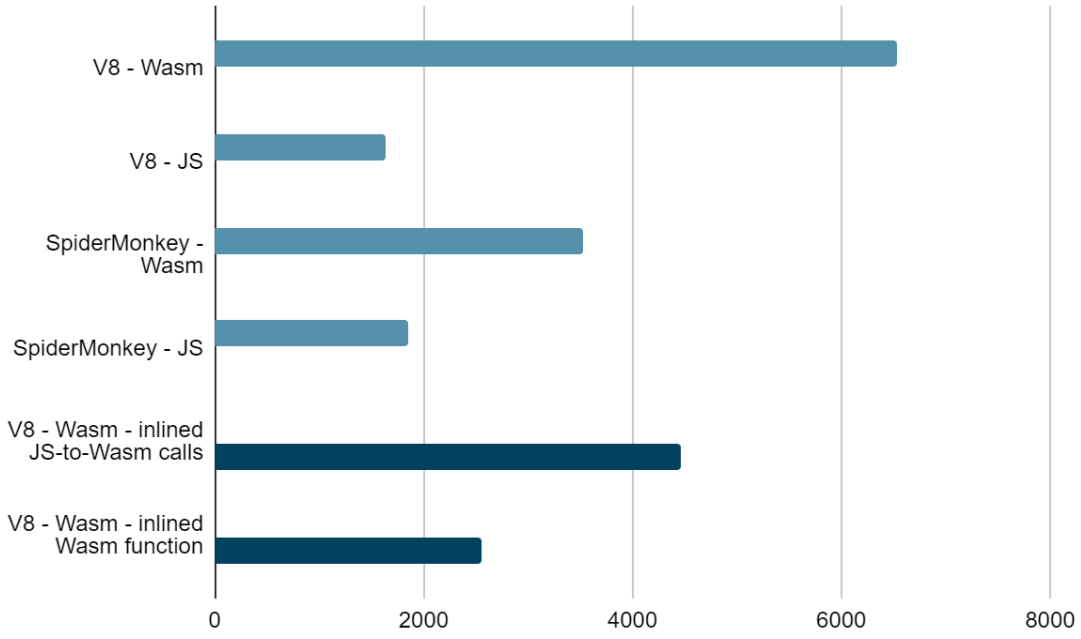
  This graph is also inlined, with a logic similar to the one provided in `JSInliner::InlineCall`.

That's all. The following phases of the TurboFan compilation run normally on a graph that now also contains nodes to generate for both the call wrapper (boxing/unboxing arguments and the result) and for the Wasm function itself.

Results

If we run the "square" test above with our prototype (CL 2609912), we see that the function becomes (almost) as fast as inlined JavaScript, and this is quite promising:

	Time [msec]
V8 - Wasm version	6540
V8 - JavaScript version	1637
SpiderMonkey - Wasm version	3526
SpiderMonkey - JavaScript version	1855
V8 - Wasm version with inlined JS->Wasm wrapper	**4461**
V8 - Wasm version with inlined Wasm function	**2561**



Finally, notice that the remaining performance gap between inlined JavaScript functions and inlined WebAssembly functions is almost entirely due to the code required to set/reset the *thread_in_wasm* flag in TLS. The example test function above - compiled by a version of TurboFan modified to not generate any code for this flag - runs exactly as fast as the version that calls the JavaScript version of “square” (1630 msec). It could be safe to omit this code for inlined functions that do not contain memory accesses.

Open issues

The main idea is quite simple, and the proof of concept in CL2609912 works for simple cases. But there are (at least!) two big issues that need to be addressed.

Trap handling

Some WebAssembly operators may ***trap*** under some conditions. In particular, out of bounds load and store memory accesses cause traps.

V8’s trap-handling mechanism is described in this doc: WebAssembly Out of Bounds Trap Handling.

In order to manage out-of-bound accesses, V8 allocates “guard pages” around the Wasm memory pages. Any access to a guard page triggers an access violation fault that may be recovered with the throwing of a JavaScript exception.

V8 provides trap-handling support for several platforms, which is then integrated with the signal handler of the embedder.

From the point of view of TurboFan, the access to Wasm memory is represented by `ProtectedLoad` and `ProtectedStore` nodes. For these nodes, the compiler generates additional data that form a “Fault location table”.

In the current implementation, for each compiled Wasm function, a `CodeProtectionInfo` struct is generated that contains the info needed to handle traps in that function.

```
struct CodeProtectionInfo {
  Address base;
  size_t size;
  size_t num_protected_instructions;
  ProtectedInstructionData instructions[1];
};
```

A `CodeProtectionInfo` contains an array of `ProtectedInstructionData` items, one for each “protected” instruction, that specifies the offset of the “landing pad” from which the exception handler should resume execution, throwing the corresponding JavaScript exception:

```
struct ProtectedInstructionData {
  // The offset of this instruction from the start of its code object.
  uint32_t instr_offset;

  // The offset of the landing pad from the start of its code object.
  uint32_t landing_offset;
};
```

All the `CodeProtectionInfo` objects are stored into a global linked list.

Fault handling is platform-dependent. On Windows an invalid memory access causes a Structured Exception Handling (SEH) access-violation exception that is handled by the SEH exception filter installed by V8 (function `HandleWasmTrap`).

When a SEH exception is raised, this filter:

1. Checks whether V8 was currently running Wasm code. The information for this is stored in TLS, in the thread-local `thread_in_wasm` flag.

2. If the fault actually happened in Wasm code, the filter iterates over the list of `CodeProtectionInfos` looking for a code range that contains the fault address. If found, it means that the fault originated in the corresponding Wasm function, and it iterates over the vector of `ProtectedInstructionData` to find the landing pad that corresponds to the fault offset, and to update accordingly the instruction pointer in the `EXCEPTION_RECORD` so that the execution will resume from there.

Trap handling for inlined Wasm functions

The trap-handling mechanism just described is broken if we inline a small Wasm function in the caller JavaScript function.
But theoretically there should be no reason why this mechanism could not be extended to also work for inlined functions. We might just need to generate `CodeProtectionInfos` also for all compiled JavaScript functions that contain inlined Wasm functions.

Instructions to set and reset the "`thread_in_wasm`" flag should still be correctly generated around the inlined Wasm code, as part of the inlining of the JS-to-Wasm wrapper.

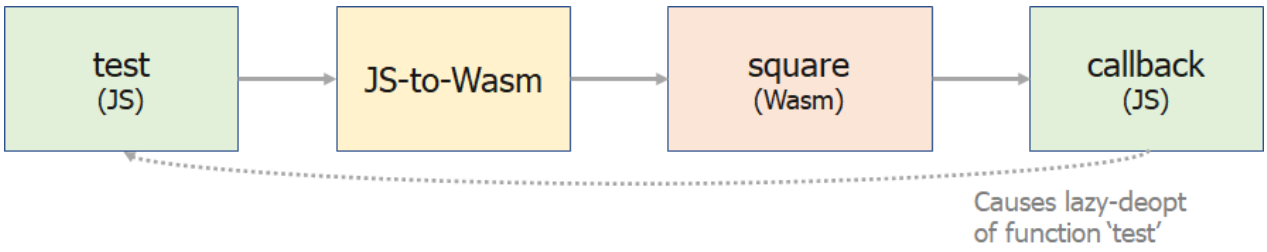
A big complication is that, while `WasmCode` does not move with garbage collections, this is not true for JavaScript `Code` objects. Therefore, in each GC we would possibly need to update the code address ranges in the global list of `CodeProtectionInfos`.

A very simple workaround is not to rely on the exception-handling mechanism to detect accesses to memory out of bounds, but to instead emit bounds-checking code. This works, but possibly with a significant loss of performance that could make inlining a little useless.

There are other issues with traps: even the mere invocation of builtin and runtime functions to trigger a trap and throw an error becomes quite complicated, as explained later in the "Passing WasmInstance to inlined code" section.

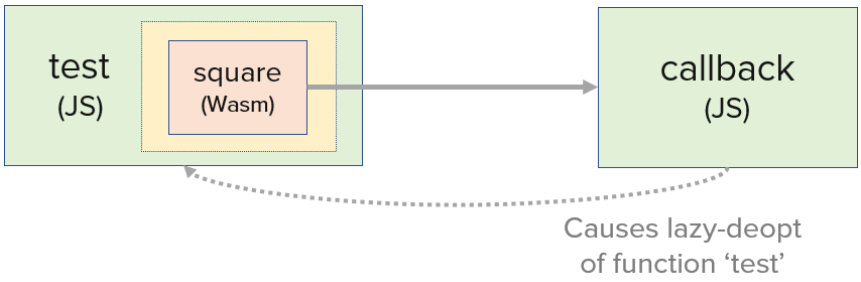
Lazy deoptimization

What happens if we have a Wasm function that calls back into a JavaScript function, which causes the lazy deoptimization of the caller function?



If we just inline the JS-to-Wasm wrapper, we can handle this case with a specific lazy-deoptimization builtin, which is called by the Deoptimizer to complete the work of the wrapper before resuming execution of the caller function in the interpreter.

But if we also inline the Wasm function, things break down.



In this case we will have a call to a Wasm-imported callback function from the inlined Wasm code. At the end of this call the compiled caller function should be deoptimized. But from where should execution resume, in the interpreter? If we resume from the JavaScript code that immediately follows the Wasm call, we skip all the code in the Wasm function after the callback.

But we cannot execute the rest of the Wasm function in an hypothetical Wasm-interpreter (which does not even exist anymore in V8). In theory, given that Liftoff compiler supports debugging, it could be possible to generate "Wasm deopt data" for inlined Wasm functions, and resume the execution in Liftoff after having suitably recreated the stack. But we do not generate any deoptimization data for Wasm code, in TurboFan, and the added complexity would probably be excessive for the usefulness of this feature.

What to do, then? Wasm inlining is especially useful for small, "leaf" functions, that generally do not call imported functions.

A possibility could then be to consider as "***non-inlinable***" any Wasm functions that:

- Calls any function imported by the Wasm module
- Might call any other "non-inlinable" Wasm function

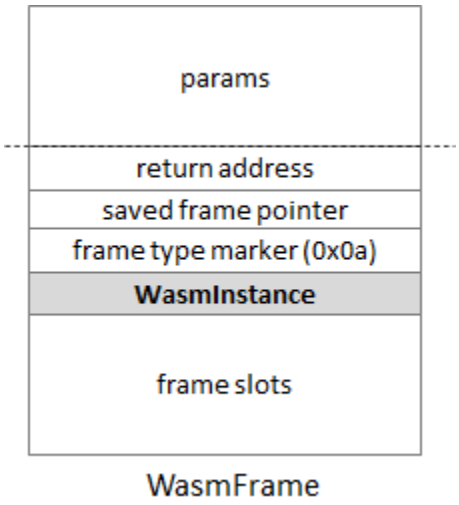
This is because the deoptimization could be triggered by any sequence of Wasm calls that ends up calling an imported function. The "non-inlinable" property of a function could be evaluated at validation or compilation time.

An additional complication is given by the "`call_indirect`" Wasm instruction, which results in an indexed call to one of the functions defined in the module Table. The simplest option is to also consider Wasm functions that make indirect_calls as "non-inlinable".

To simplify things, we could just avoid inlining any Wasm function that makes a direct or indirect call.

Passing WasmInstance to inlined code

A Wasm function sets up a specific stack frame, `WasmFrame`, which contains the WasmInstance in the fourth frame slot (see `CodeGenerator::AssembleConstructFrame`).



There are several builtins and runtime functions that expect that the WasmInstance is accessible from the stack frame. This doesn't work when we inline a Wasm function, because the JS caller function runs with a `JavaScriptFrame` (`StandardFrame`) that doesn't have this WasmInstance.

The accessors used for this are:

- `TNode<WasmInstanceObject> WasmBuiltinsAssembler::LoadInstanceFromFrame()` called by builtin functions like `WasmThrow`, `WasmTrap`, …

- `WasmInstanceObject GetWasmInstanceOnStackTop(Isolate* isolate)` called by runtime functions like `Runtime_WasmThrowCreate`, `Runtime_WasmTableGrow`, and so on.

Possible workarounds

There are no simple solutions to this issue. Possible workarounds are:

- We could store the WasmInstance not on the stack but somewhere else, for example in TLS. But we would need to change how all Wasm functions access their instance, not only the inlined ones. Also, this wouldn't work because we would need to store a whole stack of Wasm functions, given that a Wasm function can call another, possibly from a different module.
- We could add a slot for a WasmInstance also to JavaScriptFrame, making sure that it's located in the same position as in the WasmFrame. This should work, but requires considerable changes to the existing code.

- We might pass the WasmInstance as argument to all runtime functions that access it from the stack. We would pay a small performance cost, which could probably be acceptable given that many WebAssembly runtime functions are rarely invoked. An exception to this are traps, which deserve a more detailed analysis.

Managing traps without WasmInstance

Currently the code that throws trap errors uses the WasmInstance retrieved from the WasmFrame. For example, the compilation of a Wasm function that contains an `unreachable` instruction produces a call to `Builtins_ThrowWasmTrapUnreachable`, which calls `Builtins_WasmTrap(kWasmTrapUnreachable)`, which is implemented in Torque as:

```
builtin WasmTrap(error: Smi): JSAny {
  tail runtime::ThrowWasmError(LoadContextFromFrame(), error);
}
```

where `LoadContextFromFrame` reads the instance from the stack:

```
macro LoadContextFromFrame(): NativeContext {
  return LoadContextFromInstance(LoadInstanceFromFrame());
}
```

There are a dozen builtins similar to `Builtins_ThrowWasmTrapUnreachable` which all call `WasmTrap` with different error codes.
How to make this work for inlined code? We can observe that `WasmTrap` does not really need the WasmInstance, which is only used to retrieve the NativeContext. If we assume that we have access to the Context in the caller (JS) function, we can just pass it to a different builtin, designed to be used from inlined code. Something like:

```
builtin WasmTrap_Inlined(context: Context, error: Smi): JSAny {
  tail runtime::ThrowWasmError(context, error);
}
```

This solution looks promising, but there are more complications with traps.

For a Wasm function the call to `Builtins_ThrowWasmTrapXYZ` is generated as a call to a Wasm runtime stub defined in that module. Function `WasmOutOfLineTrap::GenerateCallToTrap` just encodes the stub index, that will be patched later when the code is added to the NativeModule:

```
__ near_call(static_cast<Address>(trap_id), RelocInfo::WASM_STUB_CALL);
```

This is normally done in `NativeModule::AddCodeWithCodeSpace`, which relocates the call to a near call to a runtime stub entry, which is a far jump table slot in the NativeModule.

For inlined code we should do the patching in `Code::CopyFromNoFlush`, but this requires many changes. We need to somehow store the information that that specific WAS_STUB_CALL is associated with that specific NativeModule. Furthermore, the call is currently implemented as a near-call and we calculate a 32-bit offset, which doesn't work if the instruction pointer is not from WasmCode for the same Wasm module but it's instead in a random Code object in the heap. So the way we generate the call needs to be very different for inlined code.

Conclusions

Inlining Wasm functions that rely on the presence of the WasmInstance on the stack, and especially with functions that may trap, is really complicated.
The simplest solution is just to not inline these functions. We already said that, at least in the initial implementation, for the sake of simplicity, we should avoid inlining Wasm functions that make any direct or indirect call.
We would probably also need to add all functions that might trap to the set of non-inlinable functions. This is quite a big limitation, though, given that there are several instructions that may produce a trap:

- Memory Instructions:
  - `load`, `store`, `memory.fill`, `memory.copy`, `memory.init`
- Table instructions:
  - `table.get`, `table.set`, `table.fill`, `table.copy`, `table.init`
- Control instructions:
  - `unreachable`, `call_indirect`
- Numeric instructions:
  - `idiv_u`, `idiv_s`, `irem_u`, `irem_s`, `trunc`

All that leads to an important question...

Is inlining Wasm functions really useful?

It is important to notice that only a subset of the functions in the module can be inlined. Obviously we can only inline in a JavaScript function only functions that are exported by the Wasm module, and it only makes sense to inline relatively small functions. We expect that the inlining at the level of Wasm->Wasm calls is already done by the LLVM compiler, in release builds.

There are also other constraints, we saw that we can't inline Wasm functions that make calls to imported functions, since they could cause lazy deoptimizations, and consequently we cannot inline functions that call other "non-inlinable" Wasm functions.

This raises the question: how many functions can actually be inlined in “real world” applications? To find out we analyzed a couple of large Wasm modules, *earthwasm.wasm*, which is part of Google Earth, and *dotnet.wasm*, which is part of Mono/Blazor.

	earthwasm.wasm	dotnet.wasm (3.2.0-preview3.20168)
Number of functions	56364	6354
Number of exported functions	65	61
Exported functions with bytecode size > 500	4	2
Exported functions that call imported functions or other non-inlinable functions	0	0
Exported functions that make indirect_calls	22	3
Exported functions that access memory	2	2
Inlinable functions	39	56

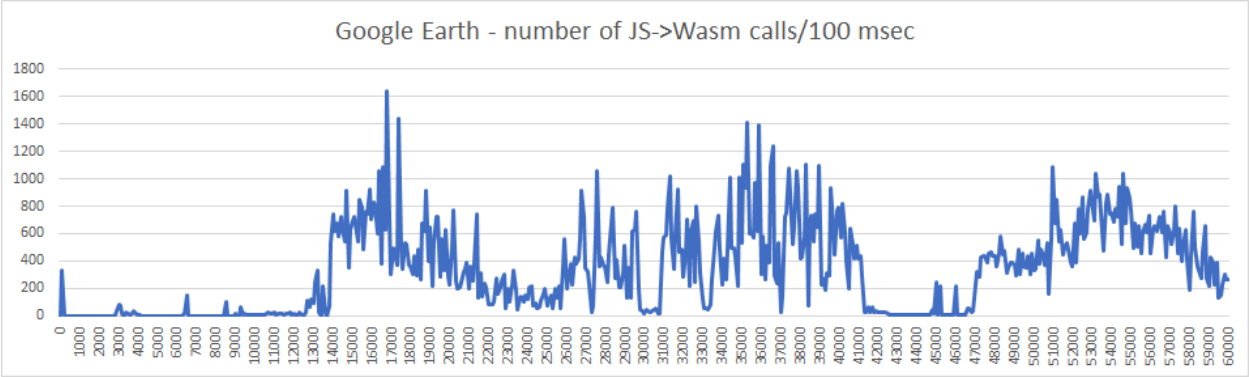
These examples tell us that the number of “inlinable” functions is relatively small, but the usefulness of inlining actually will depend on the number of times these functions are called from JavaScript.

By instrumenting V8, we can measure how many times Wasm functions are actually called by JavaScript, in a few Wasm-based real-world web applications running with Chromium:

	Peak calls/sec
Google Earth	~7500
Adobe Acrobat Demo	~1700
Blazor Demo	~1500
Doom 3 Demo	~150

Here we are only counting calls to Wasm functions that can “easily” be inlined (i.e. leaf functions that don’t access the memory).

The results vary considerably, and obviously depend on how Wasm is used by JavaScript in different apps. Here the numbers are measured in the time intervals where there is the most interaction with the user interface, which are usually the busiest intervals for what concerns the interaction with WebAssembly code.
For example, for Google Earth, we measure this trend of number of calls bucketed in 100msec intervals:



Conclusions

The original idea for the WebAssembly application model was to call Wasm from JavaScript to run long workloads, and for this kind of applications, inlining or not inlining Wasm functions will not make much difference. But there are also applications like Google Earth that are quite "chatty" and make a large number of calls to Wasm from JavaScript.

WebAssembly is quickly evolving and it is not just for games and for computationally heavy applications. It could be used, on the Web or server-side, with applications where a number of small Wasm modules are integrated in a JS system, with very frequent calls between the two. In this case the cost of the transition between WebAssembly can become significant, and making these calls as fast as possible will enable new scenarios.

References

Faster JS to WASM calls, Benedikt Meurer