Expand fast API calls with Exceptions and Re-entry to JavaScript

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

Contact: ahaas@
Contributors: dinfuehr@, mlippautz@
Status: Inception | Draft | Accepted | Done

LGTMs needed

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

Abstract

Currently fast API calls are not supported directly for API functions that can throw exceptions or call back to JavaScript. Instead, an API function can inform the caller that it wants to throw an exception, in which case the caller calls the API function a second time, but this time with the ability to throw exceptions. This “fallback” mechanism turns out to be quite expensive though. This design doc discusses ways how fast API calls can support exceptions and calls back to JavaScript without using the existing fallback method.

Objective

Support exceptions and re-entry to JS for fast API calls, so that the currently existing fallback mechanism can be removed, and so that more functions can be called with the fast API.

Value proposition to V8/Chromium

Without the fallback mechanism, fast API calls become 30%-40% faster according to micro-benchmarks. Additionally, the fallback mechanism can only be used by API functions that

can detect the need for a fallback before changing any state. Otherwise the fallback mechanism would become observable.

Background

V8's fast API calls allow direct calls from generated code to C++-written Web API functions. Because of the direct call from generated code to C++ code there is no exit frame on the stack that would help with stack handling, as it is needed to support GC and exceptions. However, many API functions have a slow path that does throw an exception, even though otherwise they would be prime candidates for the fast API calls. Therefore there exists a fallback mechanism, where the API function can tell the generated code that it could not execute correctly within the restrictions of V8's fast API calls. The generated code would then call a slow-path version of the same API function which can execute without restrictions.
Several assumptions that were made in the design of fast API calls turned out to be incorrect, see e.g. here:

- Calling directly from generated code to C++ code makes a big performance difference.
    - As it turns out, a minimal C-Entry stub only adds 3%-5% call overhead.
- Passing parameters in registers makes a big performance difference.
    - The C++ bindings code typically starts with a function call to unwrap the receiver. For this function call all register parameters get spilled to the stack anyways, so passing parameters on the stack in the first place would not have made a difference.
- The fallback mechanism is cheap.
    - As it turns out, the performance overhead of handling a potential fallback is significantly more expensive than calling through a C-Entry stub. For a no-op API function with 0 parameters, a generic API call is actually faster than a fast API call, because of the overhead of the fallback mechanism.
- The performance benefits of fast API calls comes from calling C++ code directly from generated code.
    - Measurements showed that the performance benefits of fast API calls come from avoiding the boxing and unboxing of parameters. Unboxing is especially expensive. Boxing added around 30% call overhead in a benchmark with 7 parameters. Unboxing adds 300%-400%.

Design

Independent of the design, the following challenges have to be solved:

- After the call to C++, how can we detect if an exception was thrown, so that we have to trigger stack unwinding?
- For stack iteration, there has to be some mechanism to skip C++ frames, e.g. some link from a JS-entry frame to a C-exit frame. Currently the C-Entry stub stores its Frame Pointer (FP) into the isolate, and the JS-Entry stub spills the FP of the C-Entry from the

isolate to its own stack frame. To reuse this mechanism, some FP should be stored in the isolate every time a call to C++ is happening.

Some ideas:

Idea 1: Direct call to C++

There could still be a direct call from generated code to C++, but generated code would also store its FP in the isolate before the call, and after the call it would check for an exception and trigger stack unwinding if needed.
The additional code could be generated as part of a Call operation in the macro assembler or code generator. Therefore it would not interfere with optimizations.

Pro

- No call indirection over a C-Entry stub, therefore call performance may be slightly better.

Contra

- Extra memory consumption for generated code: Extra code is generated for each call to C++ with the fast API.
- Getting the Turboshaft function doing the fast call to also catch the exception from the fast API call is quite challenging, because in the graph building code, instruction selection code, and code generation code there is are many code locations that assume that FastApiCalls cannot throw, or that calls to C functions cannot throw.

Learnings from a prototype

I wrote a prototype of this idea. In the prototype an exception gets thrown in the C++ code by allocating an exception object and storing it on the isolate. Then a `threw_exception` flag is set on the isolate, followed by a return. After the return, generated code loads the `threw_exception` flag from the isolate using the root register, and checks if it is set. If it is set, a `TriggerException` builtin is called, which contains those parts of the C-Entry strub that deals with triggering stack unwinding. Additionally I removed the check for a potential fallback.
I then tried to catch the exception in the caller of the TurboFan function that did the fast API call, which worked well. However, catching the exception in the TurboFan function itself failed, both in JavaScript and in WebAssembly.
Performance:
I measured a micro benchmark on my workstation, which calls `d8.test.FastCAPI.add_all` in a loop with 1’000’000’000 iterations. There are three configurations: a pure JavaScript version; a WebAssembly version which retrieves the receiver of the fast API call within the loop; and a WebAssembly version which retrieves the receiver once before the loop, and then uses the same receiver for all calls.

Baseline performance:
JavaScript: 12225.7ms
WebAssembly (within): 11276.9ms
WebAssembly(outside): 10726.5ms

Performance with the change:
JavaScript: 9254.3ms (-24%)
WebAssembly (within): 10566.0ms (-6%)
WebAssembly(outside): 8212.7ms (-23%)

Here are some findings:

- It is really difficult to add exception handling support to TurboFan and Turboshaft. That is, I did not manage to add support that a TurboFan function that does the fast API call also catches the exception. This seems to be a problem with the fast API in general, see this issue. Catching the exception in the caller of the TurboFan function was not a problem.
  [Update]: I did manage to make it work. I'm not 100% confident though that it would work in all cases.
  Here are some findings:
  - The instruction selector encodes in the opcode whether a call can throw an exception and should therefore show up in the handler table. However, this information cannot be encoded if the arch opcode is `kArchCallCFunction`. The bits which encode the exception information may be used differently for `kArchCallCFunction`, e.g. the number of parameters gets encoded in the opcode as well.
    - [Update]: I reduced the maximum number of FP parameters from 20 to 8, to free bits to store whether an exception is thrown. This is fine, because FP stack parameters are not supported anyways, and there are only 8 FP registers for parameters.
  - The code generator collects exception handler data in `RecordCallPosition`, but for `kArchCallCFunction` we don't call `RecordCallPosition`, we only call `RecordSafepoint`.
    - **[Update]:** I added the important lines of `RecordCallPosition` to the handling of `kArchCallCFunction` in the code generator, which works just fine.
  - For JavaScript, a `FastApiCallOp` typically does not throw, so at the moment a `FastApiCallOp` gets lowered, there is no information anymore about potential exception handlers. Furthermore, it seems like the exception handler gets eliminated in the JavaScript frontend, because there does not seem to be an instruction that can actually throw.
    - **[Update]**: I managed to make it work, but to make `FastApiCallOp` able to throw requires changes in many different places, and I'm not confident that I found all places. It would probably be better to get rid of

`FastApiCallOp`, and instead lower to a normal call immediately. I think the biggest reason for having `FastApiCallOp` was to support the fallback to the slow call, which we want to remove anyways.

- In WebAssembly, the construction of the correct Turboshaft graph was easy, all it took was using `CallAndMaybeCatchException` instead of `__ Call`.
  - **[Update]**: In a new version we check for a thrown exception in the Turboshaft graph. Thereby not the call to the C++ function can throw, but the call to the runtime function that triggers stack unwinding. For calls to runtime functions, `CallAndMaybeCatchException` is used anyways if the flag `CheckForException::kCatchInThisFrame` is passed to `CallBuiltinThroughJumptable`.
- There seem to be multiple ways to add flags to the isolate that can be accessed from generated code and from C++. I wonder if this should be unified, to have only one method.
- I did the experiment in d8, where I added a fast Api function to d8-test.cc. There it was difficult to even throw an exception, because no `Context` is available. `Isolate::GetCurrentContext` is not available, it returns an empty handle. I don't know if it should be set, and why it is not set.
  - **[Update]**: I store the context now in a second internal field in the FastAPI receiver object to have it available. This is to simulate that the context is available anyways in the fast API function. If that assumption is wrong, then we would probably have to pass the context as a parameter to fast API functions that can throw an exception.
- In WebAssembly we would have to reset the `thread_in_wasm` flag before calling the builtin to trigger stack unwinding.
  - **[Update]**: This is actually a bug in the current implementation of fast calls from WebAssembly, because it should always reset the `thread_in_wasm` flag before calling C++.
- The implementation of `CallCFunction` resets `fast_c_call_caller_pc_address` after the call to C++. However, this value is needed in the stack unwinding to find the stack frame where to start unwinding.
  - **[Update]**: Stack unwinding can be triggered by calling the runtime function `PropagateException`. This runtime function just returns the exception sentinel, the CEntry stub then triggers stack unwinding. The stack unwinding of the CEntry stub does not depend on `fast_c_call_caller_pc_address`. The exception itself is already stored in the isolate, so nothing has to be done about that.
- There is a flag, `javascript_execution_assert`, which is set during a fast API call, and prevents C code to call back into JavaScript. I forgot to reset this flag after catching an exception,

Idea 2: Introduce a C-Entry stub for fast API calls

Similar to generic API calls there would be a C-Entry stub that gets called from generated code, and that would forward the call to C++. The C-Entry stub would store an FP (either its own, or the one of the caller) in the isolate, and it would check after the call to C++ if an exception occurred. The difference to the existing C-Entry stub is the possibility of a mix of tagged and untagged parameters.
For tagged parameters the fast C-API currently allocates stack slots, and then passes a pointer to the stack slot to the C++ function, where the pointer is interpreted as a `Local<>`. It is known at compile time if a stack slot contains a tagged value, so the stack slot is considered in the existing safepoint creation.
An open question is, how should parameters be passed to the C-Entry stub, and how are parameters forwarded to the C++ function.

Alternative 1: use C calling conventions

The C-Entry stub can be called with C calling conventions, thereby parameters would already be in the right registers and in the right stack slots ato be used by the C++ function. The problem is that there would be two return addresses on the stack instead of just one if the C-Entry stub just forwards the call to C++, the return address to generated code, and the return address to the C-Entry stub. There is an easy solution though, the C-Entry stub can pop the return address to the generated code to a callee-saved register. This fixes the stack for the C++ function, and after the C++ function returns to the C-Entry stub, the return address to the generated code gets pushed back on the stack, and the C-Entry stub returns as well. By marking the callee-saved register as **caller-saved** in the CallDescriptor, it would be guaranteed that the register is unused when the C-Entry stub is called.
The C-Entry stub would not build up a stack frame. Instead it would store the FP of the caller to the isolate. The current fast API calls already do that for GC support, I don't know yet if this is good enough for exception handling and JS-re-entry.
The CallDescriptor could also be set up such that there would be a hole on the stack between the stack frame of the generated code and the parameters for the C++ function, so that the C-Entry stub could fill the hole with the setup of a stack frame. For that, however, the C-Entry stub would somehow have to be able to find the hole.

Pro

- This C-Entry stub could be enabled without changing anything on the Blink side. The C++ code in the Blink Bindings would never know that it now gets called via the C-Entry stub and not directly from generated code.
- This is probably the easiest alternative to implement. It would need a slight adjustment to the call descriptor, so that the C++ target gets passed to the C-Entry stub as a parameter.

Learnings from a prototype

I reused parts of the prototype of direct calls for a short experiment with a C-Entry stub that gets called with C calling conventions. My hope was that by calling the C-Entry stub as a builtin that can throw exceptions, the existing exception handling support would already take care of the exception handling. Throwing an exception already worked in the other prototype, so it would just work the same in this prototype, with the difference that in this prototype, stack unwinding would be triggered in the C-Entry stub directly, without calling a special builtin for that first.
I found a fundamental issue though that does not make this approach fundamentally easier than the direct call to C++.
To make the existing exception handling work out of the box, we have to call the C-Entry stub as a builtin, and not as a C++ function. Having a call to a C++ function was the fundamental problem in the other prototype. The C-calling conventions, however, require that the caller pops stack parameters. The `CallDescriptor` allows to define that parameters are passed to a builtin with C-calling conventions, but it does not allow to define who pops the parameters off the stack. So either we can add exception handling support to C calls, or we can add parameter handling support to builtin calls.

Alternative 2: provide all parameters over the stack

Passing parameters over registers seems faster intuitively. However, our analysis showed that all parameters get spilled on the stack anyways in the C++ bindings code, so writing them on the stack in generated code would not create overhead. The generated code could therefore write all parameters into a stack area, and then pass the beginning of the stack area as a pointer to C++. C++ could then access the stack area like an array.
The stack area could just be the normal stack parameter area, i.e. the call descriptor defines all parameters as stack parameters. The C-Entry stub could then just pass '`SP + 8`' to the C++ function in a parameter register. The `'+8'` is needed so that the return address is not part of the array.

Pro

- The slot in the stack area itself could be interpreted as a `Local<>` already, so there would not be a need to allocate stack slots for tagged parameters. However, in that case we would have to emit some kind of safepoint information to indicate which slots in the stack area contain tagged values. With conservative stack scanning that would naturally not be necessary.

Alternative 3: extend the existing C-Entry stub to support untagged parameters

If a mix of tagged and untagged parameters are passed to the C-Entry stub, then some kind of safepoint information would be needed by the GC to know which parameters should be

scanned. Since the C-Entry stub is generic, the caller would have to provide the safepoint information. An additional safepoint could be emitted for calls with untagged parameters. If there is no additional safepoint, the call would be a generic call to a C-Entry stub, and all parameters would be considered tagged.
As far as I know, parameters are passed to the C-Entry stub in 8 byte slots, so an unboxed float64 parameter would fit into a single slot.

Pro

- Code sharing: the existing C-Entry stub, with all the infrastructure that exists for it.

Con

- To throw an exception, the C++ function returns the exception sentinel to the C-Entry stub. With possible untagged return values, however, this mechanism is not possible anymore, because an untagged return value could match the exception sentinel by accident. Additionally, if the return value is a float64, then the C++ function has no control over the value in the GP return register `rax` at the end of the function. It is therefore unclear how exception handling would work with the existing C-Entry stub and untagged return values.

Alternative 4: use NaN boxing to pass all parameters as float64

NaN boxing would allow more efficient unboxing of Numbers. In the bindings code parameters can be checked for NaN. If they are not NaN, then the parameter is an unboxed double parameter. If they are NaN, then the parameter is either NaN, or a full pointer is encoded into the payload of the NaN. The full pointer would have to be interpreted the same as a `Local<>` at the moment, e.g. type check.

Pro

- Uniform handling of parameters
- Fast access to float64 parameters

Security considerations

- Fast API calls by themselves allow type errors to propagate from generated code into C++ implementation of API functions. This proposal neither increases nor decreases this risk.
- I don’t see a reason why supporting exceptions or calls back to JS would be any different in terms of security than the support in generic API calls.
- Fast API calls, as everything else that only exists for optimizing compilers, is a bit more difficult to test and to fuzz, simply because a function first has to be optimized before it can be tested.

Test plan

Most of this change is a refactoring of fast API calls, so the test coverage of existing tests is already good. Additionally we have to add test coverage for API functions that throw exceptions, and API functions that call back to JavaScript.
We plan to add additional fast API functions to d8-test.cc, which then get called from mjsunit tests. Fuzzers can then also pick up the new functions.
Additionally we get test coverage from the tests for API functions that can then be called with the fast API. There we have to be careful though that currently only TurboFan/Turboshaft does fast API calls, so if tests for the API functions don't use TurboFan/Turboshaft, e.g. because they don't run long enough, then we don't get test coverage from that. There seems to be a virtual test suite for web tests to test web APIs with the fast API.

Rollout considerations

The rollout would happen incrementally by landing a sequence of CLs. The CLs would include at least the following steps:

1) Introducing a new C-Entry, or a new way to do fast API calls;
2) Adjust API functions on the Blink side as needed;
3) Remove the fallback mechanism in V8;
4) Clean up the fallback code in Blink