Turbocall: the Just-in-time compiler for Deno FFI

2024-03-25, by littledivy

In this post, we will explore the lesser known optimization in Deno that makes FFI fast.

Introduction - the engine

V8 [1] Isolates are little sandboxes that run JS. JavaScript runtimes give you the ability to call native functions by reaching out of this sandbox. These native functions are often referred to as "bindings".

Optimizing these bindings are one of the most important optimizations in a JavaScript runtime. Over the years, V8 has made significant improvements in this area to make bindings faster for embedders.

Let's look at an example of a V8 C++ binding:

```
void Add(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  // Check the number of arguments passed.
  if (args.Length() < 2) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong number of arguments",
        NewStringType::kNormal).ToLocalChecked()));
    return;
  }
  // Check the argument types
  if (!args[0]->IsNumber() || !args[1]->IsNumber()) {
    isolate->ThrowException(Exception::TypeError(
        String::NewFromUtf8(isolate, "Wrong arguments",
        NewStringType::kNormal).ToLocalChecked()));
    return;
  }
  // Convert the arguments to numbers.
  double value = args[0]->NumberValue(isolate) + args[1]->NumberValue(isolate);
  // Create a new Number value and set it as the return value.
  Local<Number> num = Number::New(isolate, value);
  args.GetReturnValue().Set(num);
}
```

This does a bunch of stuff, like checking the number of arguments, type checking, converting arguments and setting the return value. Moreover, V8 has to jump through (quite literally) a lot of hoops to make this work. It sets up guards and jumps out of the optimized JIT code to the runtime.

What if there was a way to call bindings without moving out of the optimized JIT code and without all the type checks?

V8 Fast API Calls[2]

V8 Fast calls are a relatively new optimization in V8.

V8 can call our native binding directly from the optimized JIT code if we provide it with the necessary type information. The necessary typechecks happen in the compiler itself including fallback to the slow path.

```
int FastAdd(int a, int b);

// Extracts type information from the function signature
v8::CFunction fast_add = MakeV8CFunction(FastAdd);
```

This results in massive speedups for repetitve native calls from optimized JavaScript. The calls are inlined and theoretically as fast as calling a native function.

Apart from native runtime bindings, one of the most common places where this optimization is used is in FFI (Foreign Function Interface) calls.

Enter Deno FFI

```
const { symbols } = Deno.dlopen("libc.6.so", {
  open: {
    parameters: ["buffer", "i32"],
    result: "i32",
  },
});
```

`Deno.dlopen` is the API to open a dynamic library. Notice anything familiar? We are defining the number of arguments, types and the return value.

We could use this information to generate optimized native binding and give it to V8!

Turbocall[3]: a JIT for JIT

Deno created a tiny assembler (in Rust ofc) to generate optimized bindings for FFI calls based on the type information.

```
Deno.dlopen("libtest.so", {
  func: {
    parameters: ["buffer", "i32", "i32"],
    result: "i32",
  },
});
```

Turbocall generates the following bindings:

```
.arch aarch64

ldr x0, [x1, #8] ; buffer->data
mov x1, x2       ; a
mov x2, x3       ; b

moxz x8, 0
br x8            ; tailcall
```

This is simply ARM64 assembly for something like this in C:

```
int func_trampoline(void* _this, FastApiTypedArray* buffer, int a, int b) {
  return func(buffer->data, a, b);
}
```

Most notably, it generates code to properly pass JS typed arrays and arguments to the native FFI

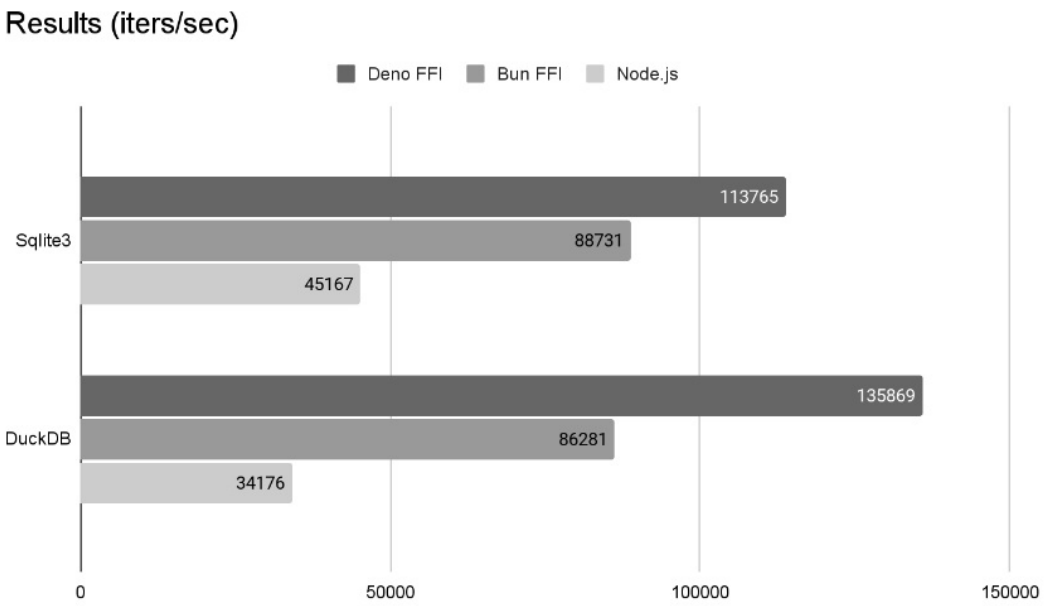
symbol.

I gave a talk on this topic at the DenoFest Meetup in Tokyo [4] which goes into more detail about the implementation.

Benchmarks

This made FFI calls 100x faster in Deno: https://github.com/denoland/deno/pull/15125

Let's see how this compares against other runtimes.



This is running sqlite3 and duckdb benchmarks on Deno, Bun and Node.js. See benchmark source. [5]

Turbocall in action

Slide from the DenoFest talk:

```
{
  parameters: ["buffer", "i32", "i32"],
  result: "i32"
}

const result = symbol(new Uint8Array(1), 10, 20);
```

```
.arch aarch64

ldr     x0, [x1, #8] ; p0: buf→data
mov     x1, x2       ; p1: int
mov     x2, x3       ; p2: int

movz    x8, 0
br      x8           ; tailcall
```

```
{
  parameters: [],
  result: "u64"
}

const result: bigint | number = symbol();
```

Typescript just works ✨

https://godbolt.org/z/jP5Wfaej6

```
sub sp, sp, 32
stp x29, x30, [sp, 16]
add x29, sp, 16

mov x0, xzr            ; _recv
ldr x19, [x1, 8]       ; result→data
movz x8, 0
blr x8                 ; symbol()

str x0, [x19]          ; copy return
ldr x19, [sp, 8]
ldp x29, x30, [sp, 16]
add sp, sp, 32
ret
```

Future

It will be interesting to see how Static Hermes[6] will compare against V8 fast calls. Both can probably generate similar code at runtime but implemented very differently.

I'm also excited about `just-js/lo`[7] which is a WIP low-level JS runtime that aims to generate V8 fast calls bindings ahead-of-time (similar to Deno) but also allow for a more engine-agnostic design where you could swap out V8 for other engines like Hermes, Quickjs.

That's it! Feel free to follow me on Twitter: https://twitter.com/undefined_void

This document is available as PDF: https://divy.work/pdf/turbocall.pdf

---

1. https://v8.dev↩
2. https://source.chromium.org/chromium/chromium/src/+/main:v8/include/v8-fast-api-calls.h↩
3. Turbocall source: https://github.com/denoland/deno/tree/ae52b49dd6edcfbb88ea39c3fcf0c0cc4b59eee7/ext/ffi↩
4. DenoFest talk https://www.youtube.com/watch?v=ssYN4rFWRIU↩
5. https://github.com/littledivy/blazing-fast-ffi-talk↩
6. https://tmikov.blogspot.com/2023/09/how-to-speed-up-micro-benchmark-300x.html↩
7. https://github.com/just-js/lo↩