How To Compile WebAssembly → C++ In A Single Pass

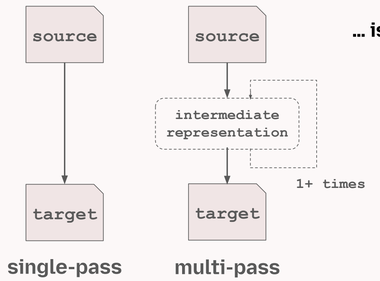
Zara Zong, Keith Winstein

What

… is WebAssembly?

WebAssembly, aka **Wasm**, is a compilation target built to run inside web browsers. Its nice properties include:

- running at *near-native speed*
- having *no undefined behavior*
- having a *fixed section ordering* (binary format) that ensures all definitions come before uses. See How for more.



… is single-pass compilation?

Single-pass compilers translate the source code *directly* into the target code. This is as opposed to multi-pass, which typically iterates through one or more intermediate representations of the source, such as an abstract syntax tree (AST).

Why

… compile from Wasm?

Wasm has *no undefined behavior*. For example, memory accesses have out-of-bounds traps, and I/O interactions must be host-provided imports.

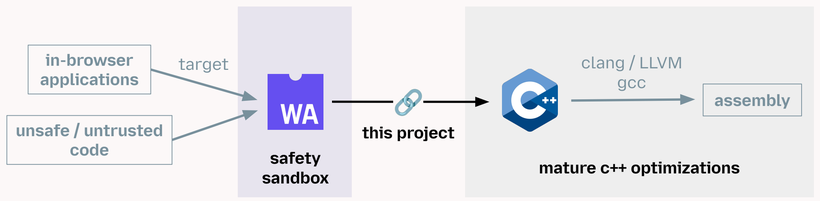
This means anything compiling *into* Wasm (e.g. untrusted code) inherits these safety guarantees, and compiling *from* Wasm preserves such guarantees in the target code.

… compile to C++?

1) To inherit *existing compiler infrastructure for optimizations*. The generated C++ can be handed to a production compiler (e.g. gcc, clang / llvm), optimized, and target many architectures.
2) C++ has convenient *language features* to express Wasm's structures. See How for more.

… single-pass?

Wasm's fixed section ordering *makes single-pass possible*, while storing minimal data (and mainly for readability of the generated code, not functionality). See How for more.



How

Introducing **wasm2c**[1,2] **!!**

… is Wasm mapped onto C++?

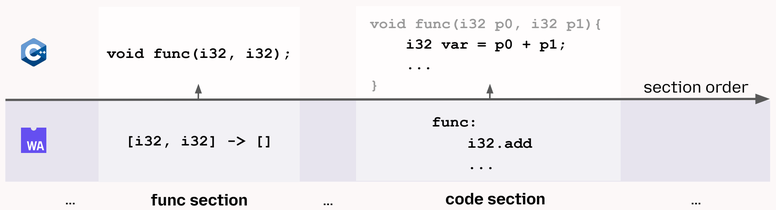
Many C++ language features make this mapping convenient. For example:

Wasm	C++
`i32, i64, f32, f64`	`uint32_t, uint64_t, float, double`
module	class
function	**member function**
global	member variable
export	public member
multiple return values	tuple

… to do so in a single pass?

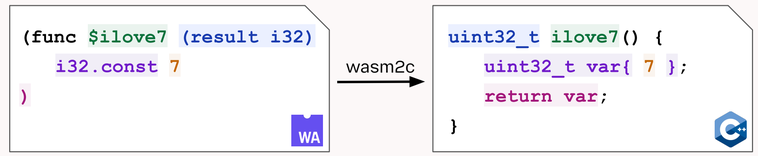
Wasm's fixed section ordering guarantees that *all definitions and dependencies come before their uses*. Thus, a single **streaming** pass is sufficient to capture the program's full meaning.

With some clever exploits of C++ structure, the resulting code can (for the most part) be incrementally generated as soon as the relevant Wasm sections stream in.



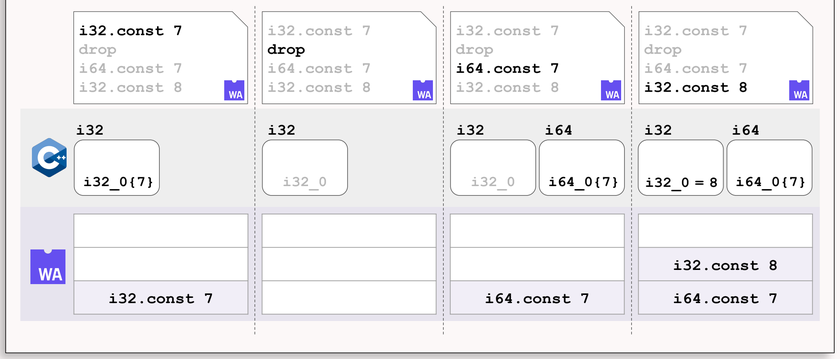
For example, Wasm's **func section** (function parameter and result types) comes before its **code section** (function content). Conveniently, C++ function *declarations* can be emitted immediately after the former, leaving *definitions* for later.

Here's a closer look inside a **function**. Note that Wasm behaves like a *typed stack machine*; assuming a semantically valid function, whatever is left on the stack at the end of a function is implicitly returned.



Not every feature is so convenient, however. The same Wasm stack slot can hold different types at different points during program execution, but a C++ variable has one fixed type for its whole lifetime.

Thus, the typedness of the Wasm stack is emulated in C++ by bookkeeping unique variable names *per type*, so each value gets a single declaration.



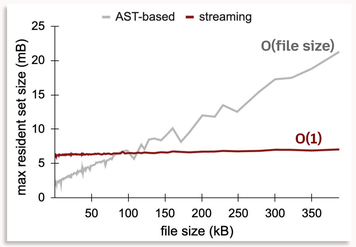
[1] This is not the first, nor the *only*, wasm2c; one widely-used implementation exists in WABT (the WebAssembly Binary Toolkit), which is written in C++, compiles to C, and builds an AST as an intermediary. However, WABT is being phased out… and as argued in the Why Stream section, there are both memory and runtime benefits to a streaming version of wasm2c.

[2] You may be wondering why the program is not named wasm2cpp or wasm2cc, since it compiles to C++ not C. Though a more literal name, the switch from C to C++ ultimately does not change the *purpose* of the program — that is, to link the safety of Wasm to the raw power of the target's toolchains, for which a program called wasm2c already exists (see footnote 1). As such, it shall inherit the name as legacy.

Why Stream[3]

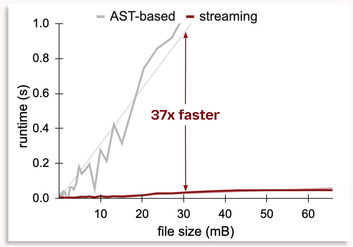
Streaming uses *less memory.*

AST construction forces memory usage to be linear with file size; streaming compiles operand-by-operand, and is thus constant.



Streaming is *faster.*

AST-based compilation of Wasm squanders the language's fixed section order, and moreover may force multiple passes.



[3] The above analyses were *not* collected on my wasm2c implementation, since it isn't built out enough. Thankfully, that is not necessary to compare AST-based with streaming compilation — existing Wasm validation programs that utilize these methods suffice. Here, I used the validators of WABT and wasm-tools respectively.

Next Up

Can we compile and run the generated C++ in-memory?

In other words, can we make a wasm2c JIT? Using Clang / LLVM APIs and virtual files, it should be possible to compile C++ → machine code without ever touching disk, then DLOpen it to run.

Can it pass the official WebAssembly Test Suite?

Only then can the correctness of this wasm2c implementation be ascertained. Pertinently, these tests may also be run in-memory.

Thank You

To Keith, Akshay, Sylvia et al, Jennifer, Greg, Haibib, and Saibi, for filling my summer with so much learning and fun.

To Michał, Yesenia, Keith (again), the many PhD and faculty mentors, and CURIS, for providing us the opportunity to test the boundless waters that is CS research.