Golang and WebAssembly

Jorge Marín

November 29, 2018

Poll time

Raise your hands if you…

- Know what Golang is (100% expected) 
- Know what LLVM is
- Know what JIT and AOT mean
- Know what Emscripten is
- Know what asm.js is
- Know what wasm is
- Know what WebAssembly is
- Prefers vim to emacs 💥

What I am going to answer

- Wat is WASM?
- Why WASM if we have Javascript?
- Why WASM if we have asm.js?
- Is WebAssembly faster than Javascript?

- **How can I run Golang code in the browser?** 

- Is WebAssembly trying to replace Javascript?
- When could WebAssembly be useful?
- When could Golang compiled to WebAssembly be useful?

Background and glossary

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

The primary sub-projects of LLVM are:

- Clang is an "LLVM native" C/C++/Objective-C compiler
- The libc++ and libc++ ABI projects provide a standard conformant and high-performance implementation of the C++ Standard Library

asm.js is a subset of JavaScript designed to allow computer software written in languages such as C to be run as web applications while maintaining performance characteristics considerably better than standard JavaScript, which is the typical language used for such applications.

Asm.js is annotated JavaScript. Designed by: Mozilla

Emscripten is a toolchain for compiling to asm.js and WebAssembly, built using LLVM, that lets you run C and C++ on the web at near-native speed without plugins.

WASM stands for WebAsembly

What is WebAssembly (aka wasm)

WebAssembly is a new type of code that can be run in modern web browsers — it is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++ with a compilation target so that they can run on the web. It is also designed to run alongside JavaScript, allowing both to work together.

WebAssembly is designed to complement and run alongside JavaScript — using the WebAssembly JavaScript APIs, you can load WebAssembly modules into a JavaScript app and share functionality between the two.

It is being developed as a web standard via the W3C WebAssembly Working Group and Community Group with active participation from engineers from Mozilla, Microsoft, Google, and Apple.

WebAssembly is a **Compiler Target** (code generated by compilers).

What is WebAssembly (aka wasm)

You can’t just execute Golang compiled programs in the browser in another machine, you need to do some compilation at the target system + make it interface nicely with web technologies.

WASM allows engineers to write their code with high level (typed and not interpreted) languages, getting all the performance gains, reliability and predictability, specifically targeting the end system.

WebAssembly is specified to be run in a safe, sandboxed execution environment. Like other web code, it will enforce the browser's same-origin and permissions policies.

You have no access to files on disks. You just have block of memory, WASM code could be called from JS and also WASM could call JS functions.

What it is not:

- **WASM is not a JavaScript killer:** WebAssembly wasn’t made to take JavaScript place. It was created to complete it where performance is critical on a web application.
- **WASM is not a new programming language:** It is worth remembering WASM is an intermediate format, binary, which works as a compiler target for other languages like C, C++, and Rust. Although a text representation exists for WASM, it’s not expected to see people programming on it as it's not expected people to code in Assembly.

History of WebAssembly

Why WASM if we have Javascript?

- JavaScript was originally intended as a lightweight language for fairly simple scripts (validating forms anyone?)
- A lot has changed since then. Modern web apps are complex computer programs, with client and server code, much of it written in JavaScript.

- But, for all the advances in the JavaScript programming language and the engines that run it, JavaScript still has inherent limitations that make it a poor fit for some scenarios. Not as fast as the operating system can run a comparable native program written in other programming languages.

Javascript -> asm.js -> WebAssembly

Why WASM if we have Javascript?

In terms of speed:
Regular Javascript < asm.js < wasm < native

A JIT compiler compiles the program as it is running, an AOT compiler compiles the program before it is running.

JIT problems:

Non-predictable performance (which we need for large, computationally-intensive apps)

Fluctuation in startups and different codes:

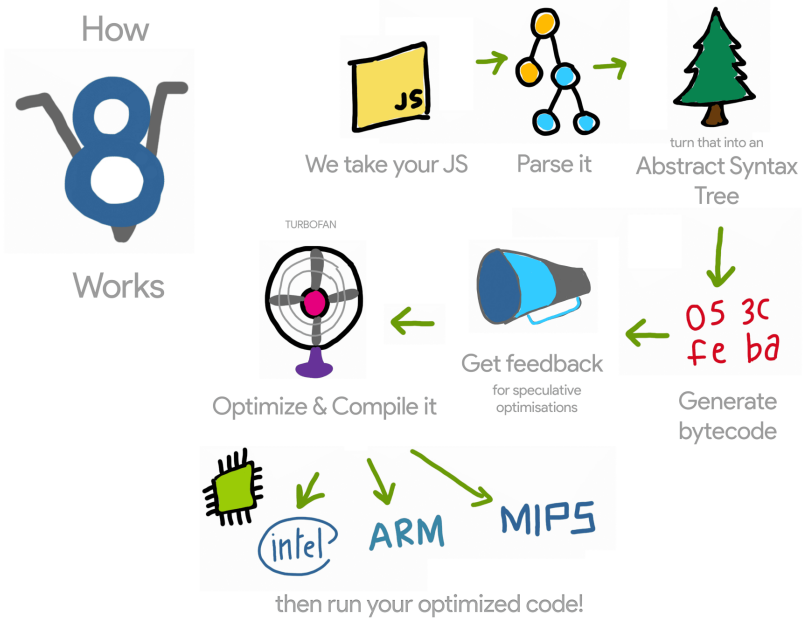
- JIT compilation time;
- as code warms up, it runs in lower-tier execution modes where it executes more slowly;
- compiler heuristics can make suboptimal choices that permanently reduce throughput;

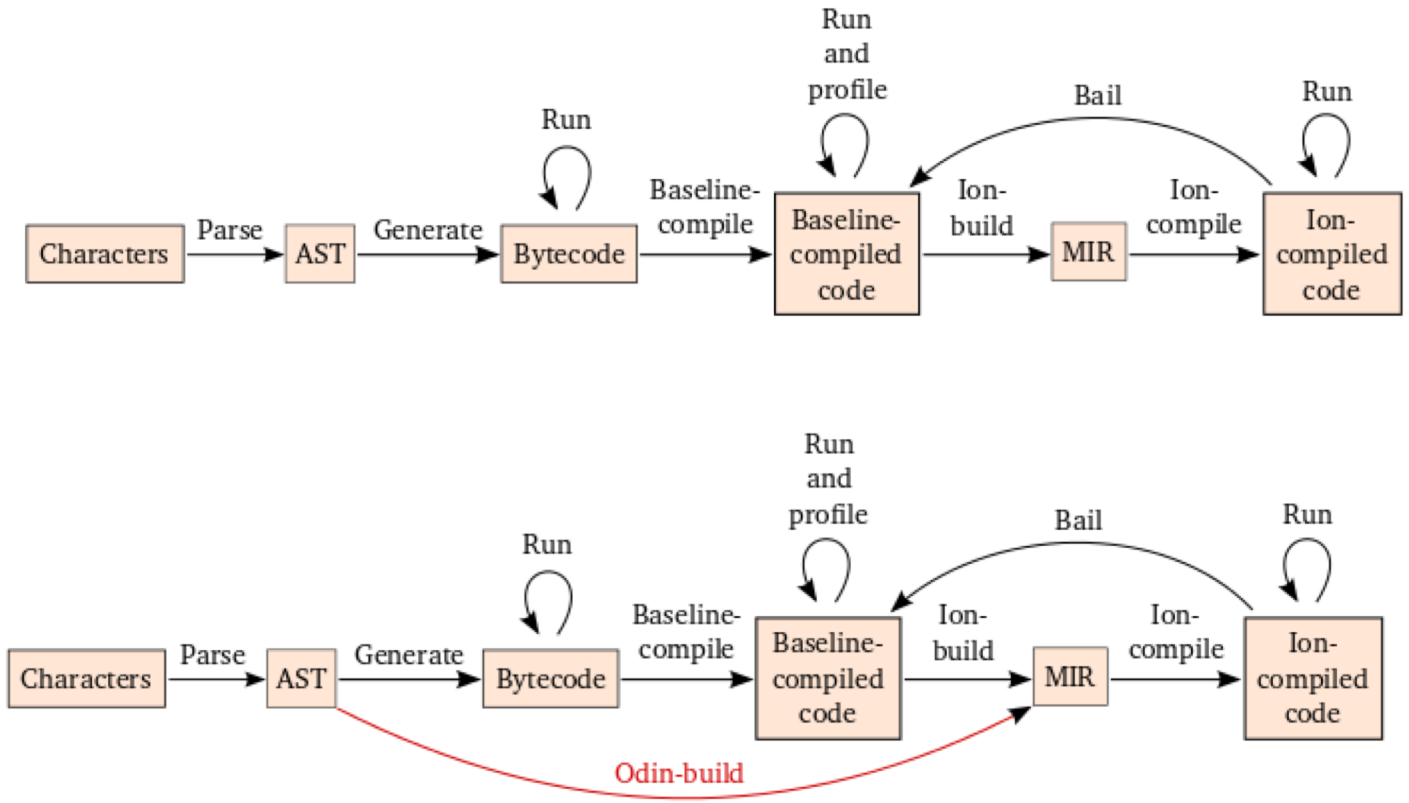
AOT problems:

- it compiles everything using the most expensive compiler without knowing if the code being compiled is hot or cold
- pathologically-large functions since these can take a really long time to compile in the top-tier compiler. With JIT compilation, the usual heuristics ensure that the top-tier compiler is never used

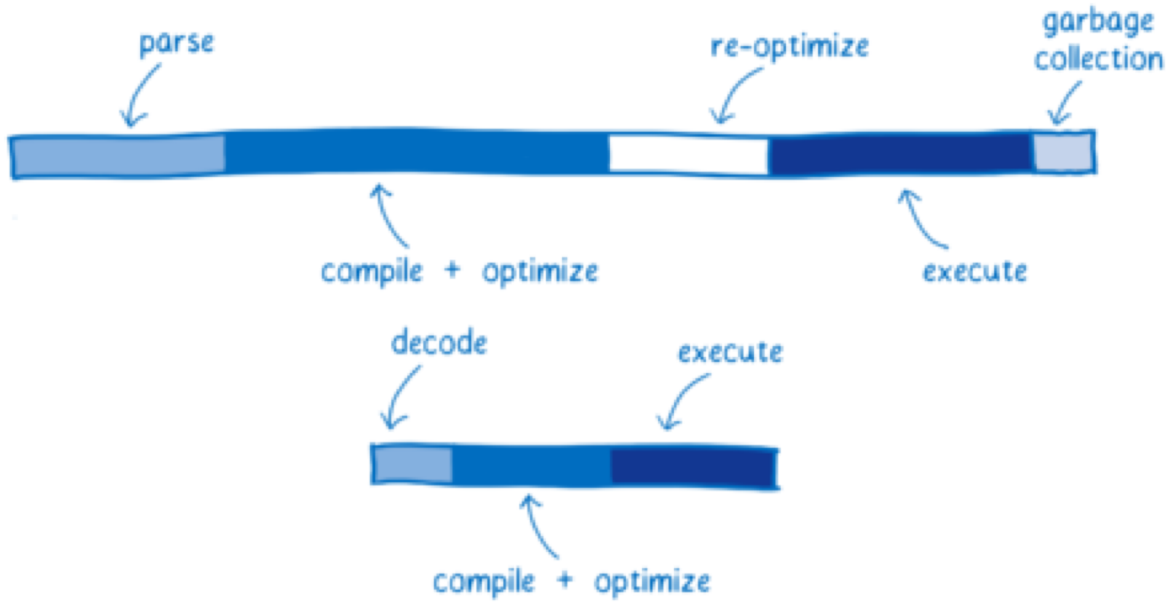
Why WASM if we have Javascript?

JIT compilation in Firefox and Chrome (interpreter and JIT compiler)





Why WASM if we have Javascript?



1. Fetching
2. Parsing
3. Compiling + Optimising
4. Reoptimising
5. Executing
6. Garbage Collection

- fetching WebAssembly takes less time because it is more compact than JavaScript
- decoding WebAssembly takes less time than parsing JavaScript
- compiling and optimizing takes less time because WebAssembly is closer to machine code than JavaScript and already has gone through optimization on the server side.
- reoptimizing doesn't need to happen because WebAssembly has types and other information built in, so the JS engine doesn't need to speculate when it optimizes the way it does with JavaScript.
- executing often takes less time because there are fewer compiler tricks and gotchas that the developer needs to know to write consistently performant code, plus WebAssembly's set of instructions are more ideal for machines.
- garbage collection is not required since the memory is managed manually.

If asm.js already overcomes some pitfalls

Why WASM if we have asm.js?

Asm.js is still Javascript. Still needs JIT.

WebAssembly binaries can be natively decoded much faster than JavaScript can be parsed (experiments show more than 20× faster)

Performance depends on the standard and not on the implementation and specific optimizations.

- Startup (bit better compressed + faster parsing)
- CPU features (64bit integers, more instructions, more to come)
- Toolchain improvements (based on lessons learned with asm.js and Emscripten)
- Predictably Good Performance
  - JavaScript -> could be made fast only using very creative methods
  - asm.js -> which could be made fast using simple methods but not all browsers did so
  - WebAssembly has more agreement upon how to optimize it.

What does this have to do with Golang? ¯_(ツ)_/¯

As we have seen, asm.js was developed with near native performance in mind. It was a middle step, using code written in C/C++ but compiled to small Javascript subset.

But you promise us Golang in the browser! Yes, then it came GopherJS, transpiling Golang code to Javascript. It works great and has great performance too.

Now, WebAssembly looks much better than asm.js and does not need to be interpreted. Golang contributors have written a WebAssembly compiler for Golang code, generating WebAssembly binaries from Go code. Awesome!

This WebAssembly binary still needs to be hooked into the page context, so a Javascript wrapper has been written while we wait for browser to proper implement a setup method.

Show me stuff

1. Compiling to wasm
2. Executing wasm with node (Litoff baseline compiler for *WebAssembly* included in V8)
3. Executing in browser
4. Adding functions
5. Evaluating from DOM
6. Manipulating DOM

Slide if everything went well

Slide if everything went wrong

Is WebAssembly faster than Javascript?

- Theory and creators claim so
- Not well-funded tests around the internet say no
  - Micro benchmarks
  - It is an MVP
  - Lots of things in the roadmap
  - JavaScript VM has had 20 years to reach its current speed
- For a more definitive answer, see the joint paper from the WebAssembly team, which outlines an expected runtime performance gain of around 30% compared to asm.js

Is WebAssembly trying to replace Javascript?

- Short answer: No! WebAssembly is designed to be a complement to, not replacement of, JavaScript.
- Long answer: While WebAssembly will, over time, allow many languages to be compiled to the Web, JavaScript has an incredible amount of momentum and will remain the single, privileged (as described above) dynamic language of the Web.

Limitations of WebAssembly and future

- Need to be compiled before hand
- Pretty new, standard WIP
- No multi-process yet

- Better compiling strategies -> more efficient
- Fixed-width SIMD (Single instruction, multiple data)
- Garbage collection

When could WebAssembly be useful?

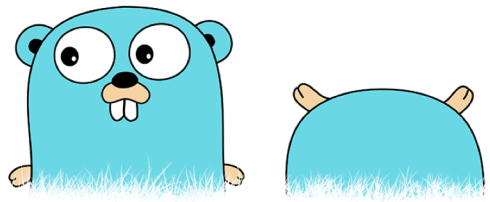
- Better execution for languages and toolkits that are currently cross-compiled to the Web
- Image/video editing
- Games (start quickly and/or heavy assets)
- Music (streaming and caching)
- VR and Augmented reality (very low latency)
- Scientific visualization and simulation.

When could Golang compiled to WebAssembly be useful?

- Your favorite language being executed in the browser
- All before + Not needing to write WebAssembly by hand
- Yes but we have GopherJS – true but we have already talked about the advantages of WASM over Javascript.

Summary

- WASM stands for WebAssembly
- Binaries that get compiled to machine code before execution or using JIT compilation
- You can get Golang in the browser with GopherJS or WebAssembly
- WebAssembly aims for near native performance
- WebAssembly at runtime is faster than asm.js or Javascript
- WebAssembly does not aim to replace Javascript
- WebAssembly is a W3C standard and a WIP (though first MVP is supported in major browsers)



Resources

- https://developer.mozilla.org/en-US/docs/WebAssembly
- https://webassembly.org/docs/faq/
- https://webassembly.org/docs/use-cases/
- https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/
- https://blog.mozilla.org/luke/2014/01/14/asm-js-aot-compilation-and-startup-performance/
- https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/
- https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8
- https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775
- https://v8.dev/blog/liftoff
- https://softwareengineering.stackexchange.com/questions/246094/understanding-the-differences-traditional-interpreter-jit-compiler-jit-interp
- https://github.com/WebAssembly/design/blob/master/FutureFeatures.md
- https://agniva.me/wasm/2018/06/18/shimmer-wasm.html
- https://tutorialedge.net/golang/go-webassembly-tutorial/
- https://webassembly.org/getting-started/js-api/
- https://blog.owulveryck.info/2018/06/08/some-notes-about-the-upcoming-webassembly-support-in-go.html
- https://github.com/sjwhitworth/golearn
- https://blog.sessionstack.com/how-javascript-works-a-comparison-with-webassembly-why-in-certain-cases-its-better-to-use-it-d80945172d79
- https://medium.com/mozilla-tech/why-webassembly-is-a-game-changer-for-the-web-and-a-source-of-pride-for-mozilla-and-firefox-dda80e4c43cb
- https://medium.com/gopherjs/surprises-in-gopherjs-performance-4a0a49b04ecd
- https://stackoverflow.com/questions/48173979/why-is-webassembly-function-almost-300-time-slower-than-same-js-function
- http://link.crwd.fr/45ZL#https://blog.acolyer.org/2017/09/18/bringing-the-web-up-to-speed-with-webassembly