Loading WebAssembly modules efficiently

By Mathias Bynens

V8 JavaScript whisperer

When working with WebAssembly, you often want to download a module, compile it, instantiate it, and then use whatever it exports in JavaScript. This post starts off with a common but suboptimal code snippet doing exactly that, discusses several possible optimizations, and eventually shows the simplest, most efficient way of running WebAssembly from JavaScript.

Note: Tools like Emscripten can produce the needed boilerplate code for you, so you don't necessarily have to code this yourself. In cases where you need fine-grained control over the loading of WebAssembly modules however, it helps to keep the following best practices in mind.

This code snippet does the complete download-compile-instantiate dance, albeit in a suboptimal way:

```
// Don't use this!
(async () => {
  const response = await fetch('fibonacci.wasm');
  const buffer = await response.arrayBuffer();
  const module = new WebAssembly.Module(buffer);
  const instance = new WebAssembly.Instance(module);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

Note how we use `new WebAssembly.Module(buffer)` to turn a response buffer into a module. This is a synchronous API, meaning it blocks the main thread until it completes. To discourage its use, Chrome disables `WebAssembly.Module` for buffers larger than 4 KB. To work around the size limit, we can use `await WebAssembly.compile(buffer)` instead:

```
(async () => {
  const response = await fetch('fibonacci.wasm');
  const buffer = await response.arrayBuffer();
  const module = await WebAssembly.compile(buffer);
  const instance = new WebAssembly.Instance(module);
  const result = instance.exports.fibonacci(42);
```

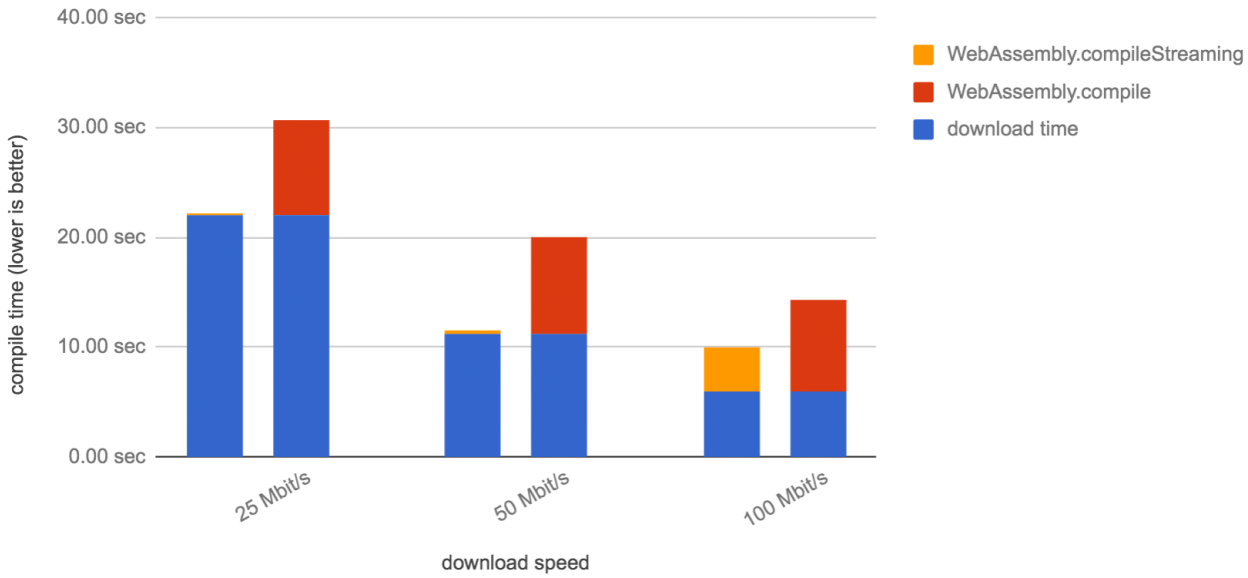
```
  console.log(result);
})();
```

`await WebAssembly.compile(buffer)` is *still* not the optimal approach, but we'll get to that in a second.

Almost every operation in the modified snippet is now asynchronous, as the use of `await` makes clear. The only exception is `new WebAssembly.Instance(module)`, which has the same 4 KB buffer size restriction in Chrome. For consistency and for the sake of keeping the main thread free, we can use the asynchronous `WebAssembly.instantiate(module)`.

```
(async () => {
  const response = await fetch('fibonacci.wasm');
  const buffer = await response.arrayBuffer();
  const module = await WebAssembly.compile(buffer);
  const instance = await WebAssembly.instantiate(module);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

Let's get back to the `compile` optimization I hinted at earlier. With streaming compilation, the browser can already start to compile the WebAssembly module while the module bytes are still downloading. Since download and compilation happen in parallel, this is faster — especially for large payloads.



To enable this optimization, use `WebAssembly.compileStreaming` instead of `WebAssembly.compile`. This change also allows us to get rid of the intermediate array buffer, since we can now pass the `Response` instance returned by `await fetch(url)` directly.

```
(async () => {
  const response = await fetch('fibonacci.wasm');
  const module = await WebAssembly.compileStreaming(response);
  const instance = await WebAssembly.instantiate(module);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

Note: The server must be configured to serve the `.wasm` file with the correct MIME type by sending the `Content-Type: application/wasm` header. In previous examples, this wasn't necessary since we were passing the response bytes as an array buffer, and so no MIME type checking took place.

The **`WebAssembly.compileStreaming`** API also accepts a promise that resolves to a **`Response`** instance. If you don't have a need for **`response`** elsewhere in your code, you can pass the promise returned by **`fetch`** directly, without explicitly **`await`**ing its result:

```
(async () => {
  const fetchPromise = fetch('fibonacci.wasm');
  const module = await WebAssembly.compileStreaming(fetchPromise);
  const instance = await WebAssembly.instantiate(module);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

If you don't need the **`fetch`** result elsewhere either, you could even pass it directly:

```
(async () => {
  const module = await WebAssembly.compileStreaming(
    fetch('fibonacci.wasm'));
  const instance = await WebAssembly.instantiate(module);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

I personally find it more readable to keep it on a separate line, though.

See how we compile the response into a module, and then instantiate it immediately? As it turns out, **`WebAssembly.instantiate`** can compile and instantiate in one go. The **`WebAssembly.instantiateStreaming`** API does this in a streaming manner:

```
(async () => {
  const fetchPromise = fetch('fibonacci.wasm');
  const { module, instance } = await WebAssembly.instantiateStreaming(fetchPromis
  // To create a new instance later:
```

```
  const otherInstance = await WebAssembly.instantiate(module);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

If you only need a single instance, there's no point in keeping the **`module`** object around, simplifying the code further:

```
// This is our recommended way of loading WebAssembly.
(async () => {
  const fetchPromise = fetch('fibonacci.wasm');
  const { instance } = await WebAssembly.instantiateStreaming(fetchPromise);
  const result = instance.exports.fibonacci(42);
  console.log(result);
})();
```

You can play around with this code example online using WebAssembly Studio.

The optimizations we applied can be summarized as follows:

- Use asynchronous APIs to avoid blocking the main thread
- Use streaming APIs to compile and instantiate WebAssembly modules more quickly
- Don't write code you don't need

Have fun with WebAssembly!

Last updated July 2, 2018.