Reality check for Cloudflare Wasm Workers and Rust

Published on: September 17, 2021

With native Rust support recently announced for Cloudflare Workers, I wanted to take a moment and write about the possibilities, but also the obstacles as a sort of reality check for myself. I'm extremely bullish about Cloudflare, Wasm, Rust, edge computing, and the recently announced native Rust support. If I'm not careful, my enthusiasm could cloud judgement, so I figured I'd take a step back and get my thoughts down on paper.

While this post is geared towards Cloudflare Workers, I'm sure some of the other edge computing platforms (eg: AWS Lambda@Edge and Fastly's Compute@Edge) also share a lot of the benefits and drawbacks.

To set the mood, I found the announcement post a bit misleading with the following quote:

> [We] baked in what you need to build small scripts or full-fledged Workers apps in Rust.

The above statement is true, but fails to mention the many caveats, so I wanted to shed some light on the rough edges, which should be expected for such new and bleeding edge technology.

For background, I created Rakaly, which is an in-browser EU4 savefile analyzer that uses Rust on both the frontend and the backend so that they can share the same high performance analysis engine. I've written about how this bet on Rust has been vindicated, allowing me to easily embed this code wherever it's needed.

Rakaly users are spread throughout the globe with a large number of users coming from every continent. In order to serve site assets in a timely manner, Cloudflare Worker Sites is leveraged to embed these assets around the world without needing an origin server. Right now, the workers are written in plain Javascript and it is embarrassing the number of times that I've gotten the API wrong (or made an even sillier mistake).

So you could say the first thing I'm excited about is being able to write a worker in a statically checked language. I'm old, I need tools to double check my work. There are other statically checked languages for workers, like the typescript workers template, but I'm not interested in juggling 15 dependencies and the associated configuration (to be fair, this most likely could be reduced to reasonable levels). As a comparison, the native Rust worker would only have one new dependency, the worker crate, as having a frontend already using Rust compiled to Wasm means the project is already tooled for Wasm.

Conflicting Message

However, the original Cloudflare Wasm announcement post should temper any rewrite in Rust mentality:

> It's important to note that Wasm is not always the right tool for the job. For lightweight tasks like redirecting a request to a different URL or checking an authorization token, sticking to pure JavaScript is probably both faster and easier than Wasm. Wasm programs

> operate in their own separate memory space, which means that it’s necessary to copy data in and out of that space in order to operate on it. Code that mostly interacts with external objects without doing any serious “number crunching” likely does not benefit from Wasm.

The above quote is from 2018, and since then there has been significant performance gains (notably startup latency) that can’t be understated. However, I’ve not seen Cloudflare communicate that using Wasm can be used for all situations, and that seems to be directly at odds with the Rust crate announcement.

For those curious, the technical reason why Wasm may not be a good choice due for performance is expounded by the tech lead behind Cloudflare workers in a thread focussed on poor Wasm performance:

> [Wasm workers] include not just your program itself, but also your programming language’s entire standard library, and all of your other transitive dependencies
>
> […]
>
> When using JavaScript, you get the standard library and all the APIs offered by the browser built-in

Makes sense. Javascript is batteries included while Wasm needs everything copied and reimplemented. I highly suggest reading the thread as it goes over additional reasons and it’s extremely insightful. The thread also contains references to significant startup improvements as they are implemented. Props to Cloudflare for willing to talk about the nitty gritty details. In the end, a bit of a bummer that choosing Wasm workers should be an exception to standard Javascript.

I wanted to create an apples-to-apples comparison between Wasm and Javascript for a Worker Site, but the Cloudflare cache API is not yet exposed in the Rust crate, and Worker Sites depend upon that API. The hope was to quantify the performance degradation of using Wasm with Cloudflare KV and Cache when serving static assets. This would let me determine the viability of a Wasm Worker Site.

So if serving static assets isn’t suitable, what are others using edge computing for?

There’s a markdown demo from Fastly, see accompanying blog post, where every keystroke sends the markdown document to the edge to be rendered into HTML. There’s less than 500 bytes of unminified client side Javascript, so the demo is remarkably lightweight. It reminds me of React server components, which accomplished the same result of offloading the markdown render to the server to reduce the client side Javascript bundle. The downside to these solutions is that an available network connection is important and at some point, more data will have been transferred to render the HTML than if the client had been given the ability to generate the HTML themselves.

But those examples focus on Javascript, let’s make a task that is a bit more CPU intensive and applicable to Rust.

Recompression

The savefiles that users upload to Rakaly are poorly compressed zip files. One can see a 2-3x improvement when the zip files are re-encoded as brotli compressed tarballs. That allows us to take advantage of the browser’s native brotli implementation on download to speed up analysis, else the Wasm unzip implementation would need to be called. Below is the code that will perform the zip to brotli tarball conversion in the browser, worker, and server. It’s simplistic for the purposes of this post.

```
fn recompress<W: Write>(zip_data: &[u8], out: W) -> anyhow:
    let cursor = Cursor::new(zip_data);
    let mut zip = zip::ZipArchive::new(cursor)
        .context("zip data was not a valid zip")?;

    let compressor = brotli::CompressorWriter::new(out, 40
    let mut archive = tar::Builder::new(compressor);
    for index in 0..zip.len() {
        let file = zip.by_index(index).unwrap();
        let mut header = tar::Header::new_gnu();
        header.set_path(file.name())?;
        header.set_cksum();
        archive.append(&header, file)?;
    }

    archive.finish()?;
    Ok(())
}
```

The portability of this code has huge ramifications. It allows us to choose where we want to run the recompression:

- Browser: Pros: Recompressing before the upload saves on bandwidth. Allows offline capabilities. Cons: Performance will vary wildly depending on the user's device. Heavier client Wasm payload.
- Edge: Pros: Offload compute heavy workload from the client and server so those can be lightweight. Cons: potentially additional costs and inflexible environment, we'll get to that last point shortly.
- Server: Pros: Most efficient implementation as native hardware is often faster. In recompression's case it executes 1.8x faster with native instructions. Cons: Uploads have to travel farther which uses bandwidth between cloudflare and the user and cloudflare and the server. May need to overallocate the server to deal with bursty loads.

In a world where best practices are always shifting on the spectrum from server rendering to full client side render, it's nice to be able to choose where to execute and not feel boxed in.

So what does our worker code look like?

```
#[event(fetch)]
pub async fn main(mut req: Request, _env: Env) -> Result<Re
    let zip_data = req.bytes().await?;

    // Optimistically guess that brotli will be 2x smaller
    let brotli_data = Vec::with_capacity(zip_data.len() /
    let mut cursor = Cursor::new(brotli_data);
    match recompress(zip_data.as_slice(), &mut cursor) {
        Ok(()) => Response::from_bytes(cursor.into_inner()
        Err(e) => Response::error(format!("{}", e), 400),
    }
}
```

I enjoy how succinct the implementation is, though we can see that the brotli output is buffered entirely in memory. Ideally we'd be able to start streaming the data back to the user as soon as we started. I'm not sure where the limitation lies. Streams exist on the web and people have created ergonomic bindings to Rust, so perhaps it is a limitation of Cloudflare's worker crate. Lack of complete coverage for a week old crate shouldn't surprise anyone. And as an aside, as far as I know, zips need to be buffered (well, trivially seekable) to access the end of the central directory record to know how to parse the zip file.

Before executing our worker, we need to hop through some additional configuration hoops. We need to first enable unbound workers on our account, as we'll be using more than 50ms of CPU time. Then we'll also update our `wrangler.toml` to specify `usage_model: unbound`. Since unbound workers are charged based on request duration, performance is important as that will save us money.

Finally we can publish and send our request. Multiple trials showed that it took 18-20 seconds (measured from inside the worker) to recompress a 7.5 MB zip file into a 3.2 MB brotli compressed tarball. Not nearly as good as the 1.4 seconds on native hardware and 2.4 seconds as Wasm in Node.js. This is still good enough to deploy, right?

Limits

Let's say I wanted to make a small tweak to get a better result at the cost of higher memory usage by tweaking the `lgwin` value from 20 to 24, which is recommended based on the file size:

```diff
- brotli::CompressorWriter::new(out, 4096, 9, 20);
+ brotli::CompressorWriter::new(out, 4096, 9, 24);
```

Locally running it saw an improvement of 10%. Attempting to execute on Cloudflare workers threw a memory exceeded error. The error description, funny enough, had an outcome of “exceededCpu” with the more correct message of about memory buried a bit deeper:

```json
{
  "outcome": "exceededCpu",
  "exceptions": [
    {
      "name": "Error",
      "message": "The script exceeded its memory limit.",
      "timestamp": 1631792399001
    }
  ],
}
```

No worries, we’ll reset `lgwin` back to 20 to be under the limit, as long as the buffered incoming zip and outgoing brotli tarball doesn’t cause the memory limit to be reached again.

Let’s say we’re still after a better compression ratio so we try to increase the brotli quality from 9 to 10. Since the increase in quality requires additional CPU horsepower, we’ll find that we exceed the worker duration limit of 30 seconds. It’s now we realize that our 20 seconds results from before is scarily close to 30 seconds especially if our input zip didn’t represent the upper bound in size. We’d most likely want to further sacrifice compression ratio by lowering brotli quality.

There’s one more limit that’ll plague us, and that is the code size limit. Worker code can’t exceed 1 MB gzipped compressed. It is deceptively easy to exceed this limit. For instance, I was considering pushing validation and data extraction of the zip from the server to the edge, as it’s another CPU intensive operation. The code doesn’t look too complex:

```rust
#[event(fetch)]
pub async fn main(mut req: Request, _env: Env) -> Result<Re
    let zip_data = req.bytes().await?;
    let zip_c = Cursor::new(zip_data.as_slice());
    let (save, _) = eu4save::Eu4Extractor::extract_save(zi
    todo!()
}
```

Unfortunately, `wrangler publish` will fail as the compressed output exceeds 1 MB. We also had to conveniently ignore that the memory limit, as that likely would be exceeded too.

There may be low hanging fruit, like tweaking `Cargo.toml`:

```toml
[profile.release]
lto = true
codegen-units = 1
opt-level = "s"
```

And in this case, we sneak under the 1 MB limit by only 9 KB, but there’s no room to grow.

In the end, we hit limits at every turn: code size, memory usage, and duration.

A Good Use Case?

At this point, one can’t be faulted for thinking Rust on Cloudflare Workers is a solution looking for a problem. Either a worker is too complex, too simple, or the worker is resource constrained, so let's double down and see if I can come up with a good use case.

The contents of these savefile zips are in a proprietary format that seems unassuming at first but have a mountain of edge cases. I've documented the text format for those interested, but it's safe to say that creating a robust parser for this syntax is difficult.

The parser I've written for this syntax is in Rust. I think it's good enough to be shared with the community, but forcing others to use Rust is a bit steep of an ask. One can create bindings to this parser in other languages (and I've created one for Javascript), but this is an arduous task simply due to the sheer number of languages.

A good middle ground (and thus use case) would be to expose this parser at the edge. The Worker can receive the proprietary syntax via the request body, and respond with a much more easily digested format like JSON. Since all languages have HTTP clients, it becomes trivial for one to take advantage of this parser in one's language of choice.

So I took the Rust code that performs the conversion to JSON and pasted it inside a worker.

```
#[event(fetch)]
pub async fn main(mut req: Request, _env: Env) -> Result<Re
    let data = req.bytes().await?;
    match jomini::TextTape::from_slice(data.as_slice()) {
        Ok(tape) => {
            let ser_tape = SerTape::new(
                &tape,
                Windows1252Encoding::new(),
                ser::DisambiguateMode::Keys,
            );
            Response::from_json(&ser_tape)
        }
        Err(e) => {
          let msg = format!("unable to parse: {}", e);
          Response::error(msg, 400)
        }
    }
}
```

And it works (and fits into 200 KB of Wasm)!

```
$ curl --data-binary "hello = yes" https://parser.rakaly.wo
{"hello":true}
```

Trying a 200 KB file works and seems performant enough for everyday use. The downside is that the savefiles we've been playing with are at least 25 MB and cause the worker to exhaust memory limits. Perhaps this becomes possible once one can stream requests and responses. And while 25 MB seems prohibitively large to upload, keep in mind these files compress extremely well. Applying brotli to the 25 MB file produces an output less than 1 MB, which is much more palatable.

One redeeming quality is that game files are much smaller, more akin to the 200 KB, so this API can still be considered useful, though significant potential remains.

Future

We've encountered many restrictions with edge computing, specifically with Wasm workers and Rust, whether it is resource limits, performance degradation, or missing APIs. The good news is that none of these issues are inherent.

I imagine that all the copying between Javascript and Wasm can be reduced once the kinks with SharedArrayBuffer are worked out.

And the missing APIs will eventually be filled in once the Rust Workers crate is given a chance to mature.

Performance will only continue to improve as Cloudflare and the V8 team iterate on solutions.

Code size will continue to be a thorny issue in the short term, unless Cloudflare raises the resource ceilings. The good news is that the far future has a glimmer of hope. The Workers Tech Lead spoke about how languages can help if they split their standard library into separate Wasm bundles so that Wasm effectively becomes batteries included.

> In order for Wasm to work really well on the edge, we need to come up with a "shared library" standard. We need each major programming language to package its standard library as a shared module, so that that one copy of that module can be loaded into all the different Workers written on that language running on the same machine.

As long as we are optimizing for code size, this could even be taken farther. Cloudflare could provide modules for popular or critical crates that don't see much churn like a hashing library or serde. If a security issue is discovered in a crate, Cloudflare could update the shared version. Of course this requires nuance, but seems a valid strategy to minimize Wasm bundles.

It also could be that Workers isn't the right level of abstraction for my use cases and that containers on the edge is what I should be interested in. Maybe then the code can take advantage of the performance of native hardware and amortize the cost of the code size. Seems like distributing a container to 200+ locations would be cost prohibited, so instead there could be a hierarchy. Regular workers distributed to all datacenters handle static assets, containers in select locations handle more resource intensive tasks, and finally a single centralized datastore.

Conclusion

Until the future arrives, we're stuck with what seems like a dearth of use cases. One must have a use case that is CPU intensive without being memory intensive, but at the same time, not too CPU intensive. And all this needs to be achieved while watching one's code weight.

I'm eagerly watching this space as I want it to succeed as edge computing enables developers like me to run a site that works well on a global scale, and I want to double down. Before doing thorough testing, I had named this post "Why I'm excited for Rust on edge".

To recap, the Rust announcement post really should have gone over these caveats and reiterated when to use so that unsuspecting developers don't have a sour experience. At least I feel more knowledgeable after my investigations. I guess I'll stick with my error prone Javascript Workers or, more likely, spend an afternoon migrating to a minimal Typescript setup.

Discuss on Hacker News

Comments

If you'd like to leave a comment, please email hi@nickb.dev