Zig And Rust

Mar 26, 2023

This post will be a bit all over the place. Several months ago, I wrote *Hard Mode Rust*, exploring an allocation-conscious style of programming. In the ensuing discussion, @jamii name-dropped TigerBeetle, a reliable, distributed, fast, and small database written in Zig in a similar style, and, well, I now find myself writing Zig full-time, after more than seven years of Rust. This post is a hand-wavy answer to the "why?" question. It is emphatically *not* a balanced and thorough comparison of the two languages. I haven't yet written my 100k lines of Zig to do that. (if you are looking for a more general "what the heck is Zig", I can recommend @jamii's post). In fact, this post is going to be less about languages, and more about styles of writing software (but pre-existing knowledge of Rust and Zig would be very helpful). Without further caveats, let's get started.

Reliable Software

To the first approximation, we all strive to write bug-free programs. But I think a closer look reveals that we don't actually care about programs being correct 100% of the time, at least in the majority of the domains. Empirically, almost every program has bugs, and yet it somehow works out OK. To pick one specific example, most programs use stack, but almost no programs understand what their stack usage is exactly, and how far they can go. When we call `malloc`, we just hope that we have enough stack space for it, we almost never check. Similarly, all Rust programs abort on OOM, and can't state their memory requirements up-front. Certainly good enough, but not perfect.

The second approximation is that we strive to balance program usefulness with the effort to develop the program. Bugs reduce usefulness a lot, and there are two styles of software engineering to deal with the:

Erlang style, where we embrace failability of both hardware and software and explicitly design programs to be resilient to partial faults.

SQLite style, where we overcome an unreliable environment at the cost of rigorous engineering.

rust-analyzer and TigerBeetle are perfect specimens of the two approaches, let me describe them.

rust-analyzer

rust-analyzer is an LSP server for the Rust programming language. By its nature, it's expansive. Great developer tools usually have a feature for every niche use-case. It also is a fast-moving open source project which has to play catch-up with the `rustc` compiler. Finally, the nature of IDE dev tooling makes availability significantly more important than correctness. An erroneous completion option would cause a smirk (if it is noticed at all), while the server crashing and all syntax highlighting turning off will be noticed immediately.

For this cluster of reasons, rust-analyzer is shifted far towards the "embrace software imperfections" side of the spectrum. rust-analyzer is designed around having bugs. All the various features are carefully compartmentalized at runtime, such that panicking code in just a single feature can't bring down the whole process. Critically, almost no code has access to any mutable state, so usage of `catch_unwind` can't lead to a rotten state.

Development process *itself* is informed by this calculus. For example, PRs with new features land when there's a reasonable certainty that the happy case works correctly. If some weird incomplete code would cause the feature to crash, that's OK. It might be even a benefit — fixing a well-reproducible bug in an isolated feature is a gateway drug to heavy contribution to rust-analyzer. Our tight weekly release schedule (and the nightly release) help to get bug fixes out there faster.

Overall, the philosophy is to maximize provided value by focusing on the common case. Edge cases become eventually correct over time.

TigerBeetle

TigerBeetle is the opposite of that.

It is a database, with domain model fixed at compile time (we currently do double-entry bookkeeping). The database is distributed, meaning that there are six TigerBeetle replicas running on different geographically and operationally isolated machines, which together implement a replicated state machine. That is, TigerBeetle replicas exchange messages to make sure every replica processes the same set of transactions, in the same order. That's a surprisingly hard problem if you allow machines to fail (the whole point of using many machines for redundancy), so we use a smart consensus algorithm (non-byzantine) for this. Traditionally, consensus algorithms assume reliable storage — data once written to disk can be always retrieved later. In reality, storage is

unreliable, nearly byzantine — a disk can return bogus data without signaling an error, and even a single such error can break consensus. TigerBeetle combats that by allowing a replica to repair its local storage using data from other replicas.

On the engineering side of things, we are building a reliable, predictable system. And predictable means *really* predictable. Rather than reigning in sources of non-determinism, we build the whole system from the ground up from a set of fully deterministic, hand crafted components. Here are some of our unconventional choices (design doc):

It's hard mode! We allocate all the memory at a startup, and there's zero allocation after that. This removes all the uncertainty about allocation.

The code is architected with brutal simplicity. As a single example, we don't use JSON, or ProtoBuf, or Cap'n'Proto for serialization. Rather, we just cast the bytes we received from the network to a desired type. The motivation here is not so much performance, as reduction of the number of moving parts. Parsing is hard, but, if you control both sides of the communication channel, you don't need to do it, you can send checksummed data as is.

We aggressively minimize all dependencies. We know exactly the system calls our system is making, because all IO is our own code (on Linux, our main production platform, we don't link libc).

There's little abstraction between components — all parts of TigerBeetle work in concert. For example, one of our core types, `Message`, is used throughout the stack:

- network receives bytes from a TCP connection directly into a `Message`
- consensus processes and sends `Messages`
- similarly, storage writes `Messages` to disk

This naturally leads to very simple and fast code. We don't need to do anything special to be zero copy — given that we allocate everything up-front, we simply don't have any extra memory to copy the data to! (A separate issue is that, arguably, you just can't treat storage as a separate black box in a fault-tolerant distributed system, because storage is also faulty).

Everything in TigerBeetle has an explicit upper-bound. There's not a thing which is *just* an `u32` — all data is checked to meet specific numeric limits at the edges of the system.

This includes `Messages`. We just upper-bound how many messages can be in-memory at the same time, and allocate precisely that amount of messages (source). Getting a new message from the message pool can't allocate and can't fail.

With all that strictness and explicitness about resources, of course we also fully externalize any IO, including time. *All* inputs are passed in explicitly, there's no ambient influences from the environment. And that means that the bulk of our testing consists of trying all possible permutations of effects of the environment. Deterministic randomized simulation is very effective at uncovering issues in real implementations of distributed systems.

What I am getting at is that TigerBeetle isn't really a normal "program" program. It strictly is a finite state machine, explicitly coded as such.

Back From The Weeds

Oh, right, Rust and Zig, the topic of the post!

I find myself often returning to the first Rust slide deck. A lot of core things are different (no longer Rust uses only the old ideas), but a lot is the same. To be a bit snarky, while Rust "is not for lone genius hackers", Zig ... kinda is. On more peaceable terms, while Rust is a language for building *modular* software, Zig is in some sense anti-modular.

It's appropriate to quote Bryan Cantrill here:

> I can write C that frees memory properly...that basically doesn't suffer from memory corruption...I can do that, because I'm controlling heaven and earth in my software. It makes it very hard to compose software. Because even if you and I both know how to write memory safe C, it's very hard for us to have an interface boundary where we can agree about who does what.

That's the core of what Rust is doing: it provides you with a language to precisely express the contracts between components, such that components can be integrated in a machine-checkable way.

Zig doesn't do that. It isn't even memory safe. My first experience writing a non-trivial Zig program went like this:

> ME: Oh wow! Do you mean I can finally *just* store a pointer to a struct's field in the struct itself?

> 30 seconds later
>
> PROGRAM: Segmentation fault.

However!
Zig *is* a much smaller language than Rust. Although you'll *have* to be able to keep the entirety of the program in your head, to control heaven and earth to not mess up resource management, doing that could be easier.

It's not true that rewriting a Rust program in Zig would make it simpler. On the contrary, I expect the result to be significantly more complex (and segfaulty). I noticed that a lot of Zig code written in "let's replace RAII with defer" style has resource-management bugs.

But it often is possible to architect the software such that there's little resource management to do (eg, allocating everything up-front, like TigerBeetle, or even at compile time, like many smaller embedded systems). It's hard — simplicity is always hard. But, if you go this way, I feel like Zig can provide substantial benefits.

Zig has just a single feature, dynamically-typed comptime, which subsumes most of the special-cased Rust machinery. It is definitely a tradeoff, instantiation-time errors are much worse for complex cases. But a lot more of the cases are simple, because there's no need for programming in the language of types. Zig is very spartan when it comes to the language. There are no closures — if you want them, you'll have to pack a wide-pointer yourself. Zig's expressiveness is aimed at producing just the right assembly, not at allowing maximally concise and abstract source code. In the words of Andrew Kelley, Zig is a DSL for emitting machine code.

Zig strongly prefers explicit resource management. A lot of Rust programs are web-servers. Most web servers have a very specific execution pattern of processing multiple independent short-lived requests concurrently. The most natural way to code this would be to give each request a dedicated bump allocator, which turns drops into no-ops and "frees" the memory at bulk after each request by resetting offset to zero. This would be pretty efficient, and would provide per-request memory profiling and limiting out of the box. I don't think any popular Rust frameworks do this — using the global allocator is convenient enough and creates a strong local optima. Zig forces you to pass the allocator in, so you might as well think about the most appropriate one!

Similarly, the standard library is very conscious about allocation, more so than Rust's. Collections are *not* parametrized by an allocator, like in C++ or (future) Rust. Rather,

an allocator is passed in explicitly to every method which actually needs to allocate. This is *Call Site Dependency Injection*, and it is more flexible. For example in TigerBeetle we need a couple of hash maps. These maps are sized at a startup time to hold just the right number of elements, and are never resized. So we pass an allocator to `init` method, but we don't pass it to the event loop. We get to both use the standard hash-map, and to feel confident that there's no way we can allocate in the actual event loop, because it doesn't have access to an allocator.

Wishlist

Finally, my wishlist for Zig.

First, I think Zig's strength lies strictly in the realm of writing "perfect" systems software. It is a relatively thin slice of the market, but it is important. One of the problems with Rust is that we don't have a reliability-oriented high-level programming language with a good quality of implementation (modern ML, if you will). This is a blessing for Rust, because it makes its niche bigger, increasing the amount of community momentum behind the language. This is also a curse, because a bigger niche makes it harder to maintain focus. For Zig, Rust already plays this role of "modern ML", which creates bigger pressure to specialize.

Second, my biggest worry about Zig is its semantics around aliasing, provenance, mutability and self-reference ball of problems. I don't worry all that much about this creating "iterator invalidation" style of UB. TigerBeetle runs in `-DReleaseSafe`, which mostly solves spatial memory safety, it doesn't really do dynamic memory allocation, which unasks the question about temporal memory safety, and it has a very thorough fuzzer-driven test suite, which squashes the remaining bugs. I do worry about the semantics of the language itself. My current understanding is that, to correctly compile a C-like low-level language, one really needs to nail down semantics of pointers. I am not sure "portable assembly" is really a thing: it is possible to create a compiler which does little optimization and "works as expected" most of the time, but I am doubtful that it's possible to correctly describe the behavior of such a compiler. If you start asking questions about what are pointers, and what is memory, you end up in a fairly complicated land, where bytes are poison. Rust tries to define that precisely, but writing code which abides by the Rust rules without a borrow-checker isn't really possible — the rules are too subtle. Zig's implementation today is *very* fuzzy around potentially aliased pointers, copies of structs with interior-pointers and the like. I wish that Zig had a clear answer to what the desired semantics is.

Third, IDE support. I've written about that before on this blog. As of today, developing Zig is quite pleasant — the language server is pretty spartan, but already is quite helpful, and for the rest, Zig is exceptionally greppable. But, with the lazy compilation model and the absence of out-of-the-language meta programming, I feel like Zig could be more ambitious here. To position itself well for the future in terms of IDE support, I think it would be nice if the compiler gets the basic data model for IDE use-case. That is, there should be an API to create a persistent analyzer process, which ingests a stream of code edits, and produces a continuously updated model of the code without explicit compilation requests. The model can be very simple, just "give me an AST of this file inn this point in time" would do — all the fancy IDE features can be filled in later. What matters is a shape of data flow through the compiler — not an edit-compile cycle, but rather a continuously updated view of the world.

Fourth, one of the values of Zig which resonates with me a lot is a preference for low-dependency, self-contained processes. Ideally, you get yourself a `./zig` binary, and go from there. The preference, at this time of changes, is to bundle a particular version of `./zig` with a project, instead of using a system-wide `zig`. There are two aspects that could be better.

"Getting yourself a Zig" is a finicky problem, because it requires bootstrapping. To do that, you need to run some code that will download the binary for your platform, but each platform has its own way to "run code". I wish that Zig provided a blessed set of scripts, `get_zig.sh`, `get_zig.bat`, etc (or maybe a small actually portable binary?), which projects could just vendor, so that the contribution experience becomes fully project-local and self-contained:

```
$ ./get_zig.sh
$ ./zig build
```

Once you have `./zig`, you can use that to drive the *rest* of the automation. You already can `./zig build` to drive the build, but there's more to software than just building. There's always a long tail of small things which traditionally get solved with a pile of platform-dependent bash scripts. I wish that Zig pushed the users harder towards specifying all that automation in Zig. A picture is worth a thousand words, so

```
# BAD: dependency on the OS
$ ./scripts/deploy.sh --port 92

# OK: no dependency, but a mouthful to type
$ ./zig build task -- deploy --port 92

```

```
# Would be GREAT:
$ ./zig do deploy --port 92
```

Attempting to summarize,

- Rust is about compositional safety, it's a more scalable language than Scala.
- Zig is about perfection. It is a very sharp, dangerous, but, ultimately, more flexible tool.

Discussion on /r/Zig and /r/rust.