How to minimize Rust binary size 📦 https://github.com/johnthagen/min-sized-rust

MIT license

9.7k stars **243** forks **109** watching Branches Activity
Tags

Public repository

1 Branch 0 Tags Go to file t Go to file Add file Code

FooIbar Fix regression of switching to new `panic=immediate-abort` flag (#87)

24233f9 · 5 months ago

.github/workflows	Fix regression of switching to new p...	5 months ago
build_std	Drop Rust 1.28 and 1.30 from CI (#5...	4 years ago
no_main	Add support for Windows `#![no_ma...`	2 years ago
no_std	Add Windows no_std project examp...	2 years ago
src	Remove setting system allocator as...	7 years ago
.editorconfig	Add editorconfig	4 years ago
.gitattributes	Initial commit of minimized Hello W...	8 years ago
.gitignore	Initial commit of minimized Hello W...	8 years ago
Cargo.lock	Add MIT license	6 years ago
Cargo.toml	Add support for Windows `#![no_ma...`	2 years ago
LICENSE.txt	Update copyright year	4 years ago
README.md	Fix regression of switching to new p...	5 months ago

README MIT license

Minimizing Rust Binary Size

build passing

This repository demonstrates how to minimize the size of a Rust binary.

By default, Rust optimizes for execution speed, compilation speed, and ease of debugging rather than binary size, since for the vast majority of applications this is ideal. But for situations where a developer wants to optimize for binary size instead, Rust provides mechanisms to accomplish this.

Build in Release Mode

Minimum Rust Version 1.0

By default, `cargo build` builds the Rust binary in debug mode. Debug mode disables many optimizations, which helps debuggers (and IDEs that run them) provide a better debugging experience. Debug binaries can be 30% or more larger than release binaries.

To minimize binary size, build in release mode:

```
$ cargo build --release
```

`strip` Symbols from Binary

OS *nix Minimum Rust Version 1.59

By default on Linux and macOS, symbol information is included in the compiled `.elf` file. This information is not needed to properly execute the binary.

Cargo can be configured to automatically `strip` binaries. Modify `Cargo.toml` in this way:

```
[profile.release]
strip = true  # Automatically strip symbols from the binary.
```

Prior to Rust 1.59, run `strip` directly on the `.elf` file instead:

```
$ strip target/release/min-sized-rust
```

Optimize For Size

Minimum Rust Version 1.28

Cargo defaults its optimization level to `3` for release builds, which optimizes the binary for **speed**. To instruct Cargo to optimize for minimal binary **size**, use the `z` optimization level in `Cargo.toml` :

```
[profile.release]
opt-level = "z"  # Optimize for size.
```

> Note
>
> In some cases the `"s"` level may result in a smaller binary than `"z"` , as explained in the `opt-level` documentation:
>
> It is recommended to experiment with different levels to find the right balance for your project. There may be surprising results, such as ... the `"s"` and `"z"` levels not being necessarily smaller.

Enable Link Time Optimization (LTO)

Minimum Rust Version 1.0

By default, Cargo instructs compilation units to be compiled and optimized in isolation. LTO instructs the linker to optimize at the link stage. This can, for example, remove dead code and often times reduces binary size.

Enable LTO in `Cargo.toml`:

```
[profile.release]
lto = true
```

Dynamic Linking: Why It Doesn't Work

Minimum Rust Version 1.0

Some might suggest using `prefer-dynamic` for smaller binaries, but this approach has critical limitations:

- **No stable ABI** - binaries break between Rust versions
- **Deployment complexity** - requires exact library matches
- **Community consensus** - static linking preferred for reliability

Reduce Parallel Code Generation Units to Increase Optimization

By default, Cargo specifies 16 parallel codegen units for release builds. This improves compile times, but prevents some optimizations.

Set this to `1` in `Cargo.toml` to allow for maximum size reduction optimizations:

```
[profile.release]
codegen-units = 1
```

Abort on Panic

Minimum Rust Version 1.10

> Important
>
> Up to this point, the features discussed to reduce binary size did not have an impact on the behaviour of the program (only its execution speed). This feature does have an impact on behavior.

By default, when Rust code encounters a situation when it must call `panic!()`, it unwinds the stack and produces a helpful backtrace. The unwinding code, however, does require extra binary size. `rustc` can be instructed to abort immediately rather than unwind, which removes the need for this extra unwinding code.

Enable this in `Cargo.toml`:

```
[profile.release]
panic = "abort"
```

Remove Location Details

Minimum Rust Version nightly

By default, Rust includes file, line, and column information for `panic!()` and `[track_caller]` to provide more useful traceback information. This information requires space in the binary and thus increases the size of the compiled binaries.

To remove this file, line, and column information, use the unstable `rustc -Zlocation-detail` flag:

```
$ RUSTFLAGS="-Zlocation-detail=none" cargo +nightly build --release
```

Remove `fmt::Debug`

Minimum Rust Version nightly

With the `-Zfmt-debug` flag you can turn `#[derive(Debug)]` and `{:?}` formatting into no-ops. This will ruin output of `dbg!()`, `assert!()`, `unwrap()`, etc., and may break code that unwisely relies on the debug formatting, but it will remove derived `fmt` functions and their strings.

```
$ RUSTFLAGS="-Zfmt-debug=none" cargo +nightly build --release
```

Optimize `libstd` with `build-std`

Minimum Rust Version nightly

> **Note**
>
> See also Xargo, the predecessor to `build-std`. Xargo is currently in maintenance status.

> **Note**
>
> Example project is located in the `build_std` folder.

Rust ships pre-built copies of the standard library (`libstd`) with its toolchains. This means that developers don't need to build `libstd` every time they build their applications. `libstd` is statically linked into the binary instead.

While this is very convenient there are several drawbacks if a developer is trying to aggressively optimize for size.

1. The prebuilt `libstd` is optimized for speed, not size.
2. It's not possible to remove portions of `libstd` that are not used in a particular application (e.g. LTO and panic behaviour).

This is where `build-std` comes in. The `build-std` feature is able to compile `libstd` with your application from the source. It does this with the `rust-src` component that `rustup` conveniently provides.

Install the appropriate toolchain and the `rust-src` component:

```
$ rustup toolchain install nightly
$ rustup component add rust-src --toolchain nightly
```

Build using `build-std`:

```
# Find your host's target triple.
$ rustc -vV
...
host: x86_64-apple-darwin

# Use that target triple when building with build-std.
# Add the =std,panic_abort to the option to make panic = "abort" Cargo.toml option work.
# See: https://github.com/rust-lang/wg-cargo-std-aware/issues/56
$ RUSTFLAGS="-Zlocation-detail=none -Zfmt-debug=none" cargo +nightly build \
  -Z build-std=std,panic_abort \
  -Z build-std-features="optimize_for_size" \
  --target x86_64-apple-darwin --release
```

The `optimize_for_size` flag provides a hint to `libstd` that it should try to use algorithms optimized for binary size. More information about it can be found in the tracking issue.

On macOS, the final stripped binary size is reduced to 51KB.

Remove `panic` String Formatting with `panic=immediate-abort`

Minimum Rust Version nightly

Even if `panic = "abort"` is specified in `Cargo.toml`, `rustc` will still include panic strings and formatting code in final binary by default. An unstable `panic=immediate-abort` feature has been merged into the `nightly` `rustc` compiler to address this.

To use this, repeat the instructions above to use `build-std`, but also pass `-Zunstable-options -Cpanic=immediate-abort` and `-Z build-std-features=` (which will disable the default `backtrace` and `panic-unwind` features) to `rustc`.

```
$ RUSTFLAGS="-Zunstable-options -Cpanic=immediate-abort" cargo +nightly build \
    -Z build-std=std,panic_abort \
    -Z build-std-features= \
    --target x86_64-apple-darwin --release
```

On macOS, the final stripped binary size is reduced to 30KB.

Remove `core::fmt` with `#![no_main]` and Careful Usage of `libstd`

Minimum Rust Version nightly

> **Note**
>
> Example projects are located in the `no_main` folder.

Up until this point, we haven't restricted what utilities we used from `libstd`. In this section we will restrict our usage of `libstd` in order to reduce binary size further.

If you want an executable smaller than 20 kilobytes, Rust's string formatting code, `core::fmt` must be removed. `panic=immediate-abort` only removes some usages of this code. There is a lot of other code that uses formatting in some cases. That includes Rust's "pre-main" code in `libstd`.

By using a C entry point (by adding the `#![no_main]` attribute) , managing stdio manually, and carefully analyzing which chunks of code you or your dependencies include, you can sometimes make use of `libstd` while avoiding bloated `core::fmt` .

Expect the code to be hacky and unportable, with more `unsafe{}` s than usual. It feels like `no_std` , but with `libstd` .

Start with an empty executable, ensure `xargo bloat --release --target=...` contains no `core::fmt` or something about padding. Add (uncomment) a little bit. See that `xargo bloat` now reports drastically more. Review source code that you've just added. Probably some external crate or a new `libstd` function is used. Recurse into that with your review process (it requires `[replace]` Cargo dependencies and maybe digging in `libstd`), find out why it weighs more than it should. Choose alternative way or patch dependencies to avoid unnecessary features. Uncomment a bit more of your code, debug exploded size with `xargo bloat` and so on.

On macOS, the final stripped binary is reduced to 8KB.

Removing `libstd` with `#![no_std]`

> **Note**
>
> Example projects are located in the `no_std` folder.

Up until this point, our application was using the Rust standard library, `libstd` . `libstd` provides many convenient, well tested cross-platform APIs and data types. But if a user wants to reduce binary size to an equivalent C program size, it is possible to depend only on `libc` .

It's important to understand that there are many drawbacks to this approach. For one, you'll likely need to write a lot of `unsafe` code and lose access to a majority of Rust crates that depend on `libstd` . Nevertheless, it is one (albeit extreme) option to reducing binary size.

A `strip` ed binary built this way is around 8KB.

```rust
#![no_std]
#![no_main]

extern crate libc;

#[no_mangle]
pub extern "C" fn main(_argc: isize, _argv: *const *const u8) -> isize {
    // Since we are passing a C string the final null character is mandatory.
    const HELLO: &'static str = "Hello, world!\n\0";
    unsafe {
        libc::printf(HELLO.as_ptr() as *const _);
    }
    0
}

#[panic_handler]
fn my_panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}
```

Compress the binary

> **Note**

> Up until this point, all size-reducing techniques were Rust-specific. This section describes a language-agnostic binary packing tool that is an option to reduce binary size further.

UPX is a powerful tool for creating a self-contained, compressed binary with no addition runtime requirements. It claims to typically reduce binary size by 50-70%, but the actual result depends on your executable.

```
$ upx --best --lzma target/release/min-sized-rust
```

> ⚠ Warning
>
> There have been times that UPX-packed binaries have flagged heuristic-based antivirus software because malware often uses UPX.

Tools

- `cargo-bloat` - Find out what takes most of the space in your executable.
- `cargo-llvm-lines` - Measure the number and size of instantiations of each generic function, indicating which parts of your code offer the highest leverage in improving compilation metrics.
- `cargo-unused-features` - Find and prune enabled but potentially unused feature flags from your project.
- `momo` - `proc_macro` crate to help keeping the code footprint of generic methods in check.
- Twiggy - A code size profiler for Wasm.

Containers

Sometimes it's advantageous to deploy Rust into containers (e.g. Docker). There are several great existing resources to help create minimum sized container images that run Rust binaries.

- Official `rust:alpine` image
- mini-docker-rust
- muslrust
- docker-slim - Minify Docker images
- dive - A tool for exploring a container image and discovering ways to shrink the size of the image.
- distroless - 2MB base image to run statically linked Rust program

References

- 151-byte static Linux binary in Rust - 2015
- Why is a Rust executable large? - 2016
- Tiny Rocket - 2018
- Formatting is Unreasonably Expensive for Embedded Rust - 2019
- Tiny Windows executable in Rust - 2019
- Making a really tiny WebAssembly graphics demos - 2019
- Reducing the size of the Rust GStreamer plugin - 2020
- Optimizing Rust Binary Size - 2020
- Minimizing Mender-Rust - 2020
- Optimize Rust binaries size with cargo and Semver - 2021
- Tighten rust's belt: shrinking embedded Rust binaries - 2022

- Avoiding allocations in Rust to shrink Wasm modules - 2022
- A very small Rust binary indeed - 2022
- The dark side of inlining and monomorphization - 2023
- Making Rust binaries smaller by default - 2024
- Tock Binary Size - 2024
- Trimming down a rust binary in half - 2024
- Reducing WASM binary size: lessons from building a web terminal - 2024
- `min-sized-rust-windows` - Windows-specific tricks to reduce binary size
- Shrinking `.wasm` Code Size

Organizations

- wg-binary-size: Working group for improving the size of Rust programs and libraries.

Legacy Techniques

The following techniques are no longer relevant for modern Rust development, but may apply to older versions of Rust and are maintained for historical purposes.

Remove Jemalloc

> **Important**
>
> As of Rust 1.32, `jemalloc` is removed by default. **If using Rust 1.32 or newer, no action is needed to reduce binary size regarding this feature**.

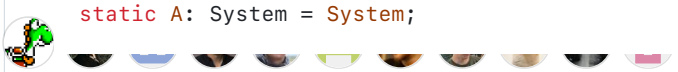
Prior to Rust 1.32, to improve performance on some platforms Rust bundled jemalloc, an allocator that often outperforms the default system allocator. Bundling jemalloc added around 200KB to the resulting binary, however.

To remove `jemalloc` on Rust 1.28 - Rust 1.31, add this code to the top of `main.rs`:

```
use std::alloc::System;

#[global_allocator]
static A: System = System;
```

Con



Languages

- **Rust** 100.0%