



TECHNISCHE
UNIVERSITÄT
WIEN



Forsyte and Friends Rust Tutorial

Florian Sextl, Mark Chimes, Adrian Rebola
2025-03-25

Should you use Rust/would you want to use Rust?

Should you use Rust/would you want to use Rust?

Simple Answer

That depends on a lot of things.

Performance

Performance

- Studies often compare to C

Performance

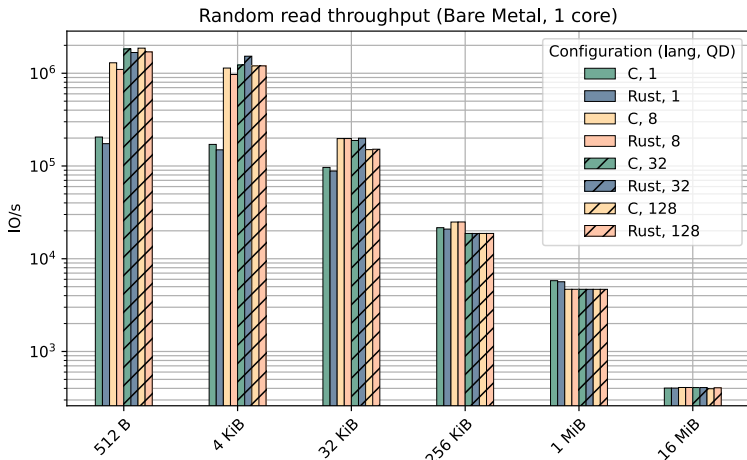
- Studies often compare to C
- Overall slower with runtime checks, similar optimized

Performance

- Studies often compare to C
- Overall slower with runtime checks, similar optimized
- optimizations due to safety, compile-time checks, ZSTs

Performance

Example: Rust for Linux NVMe driver



Performance

Further examples:

- sudo-rs
- fish
- zlib-rs
- skrifa (new font engine in Chrome)
- at AWS (e.g. Firecracker)
- Linux/Windows/Android/...
- and many more

Safety Guarantees

Safety Guarantees

Memory Safety

Safe from: use-after-free, null pointer dereference, out-of-bounds access, uninitialized memory, iterator invalidation, . . . due to type/ownership system

Safety Guarantees

Memory Safety

Safe from: use-after-free, null pointer dereference, out-of-bounds access, uninitialized memory, iterator invalidation, ... due to type/ownership system

Thread Safety

Safe from data races due to type/ownership system and `std` thread APIs

Safety Guarantees

Memory Safety

Safe from: use-after-free, null pointer dereference, out-of-bounds access, uninitialized memory, iterator invalidation, ... due to type/ownership system

Thread Safety

Safe from data races due to type/ownership system and `std` thread APIs

Basic Rule

Safe Rust can never directly cause Undefined Behavior!

Safety Encapsulation via `unsafe`

Safety Encapsulation via `unsafe`

unsafe Block

- allows `unsafe` “super powers” in a limited scope
- enclosed, local safety reasoning

```
let slice = unsafe {  
    // SAFETY: raw_ptr is guaranteed to be non-null  
    // and will point to n elements in memory.  
    from_raw_parts(raw_ptr, n)  
};
```

Safety Encapsulation via `unsafe`

`unsafe` Block

- allows `unsafe` “super powers” in a limited scope
- enclosed, local safety reasoning

`unsafe` Functions

- denotes uncheckable invariants/requirements
- also for FFI functions

Safety Encapsulation via `unsafe`

`unsafe` Block

- allows `unsafe` “super powers” in a limited scope
- enclosed, local safety reasoning

`unsafe` Functions

- denotes uncheckable invariants/requirements
- also for FFI functions

Library Soundness

A call to a safe API must never cause Undefined Behavior!

Type System

Type System

High level, compile-time abstractions

```
/** Required state: SAT
 *   State after:      SAT */
IPASIR_API int32_t ipasir_val (void * solver, int32_t lit);

pub fn value(self: &Solver<SATState>, literal: Literal) ->
    Result<Assignment, Error>;
pub fn checked_sat(self: Solver<SolvedState>) ->
    Result<Solver<SATState>, Error>;
```

Type System

Type-based error handling

```
pub fn solve_formula<F, C, L>(formula: F) ->  
    Result<Option<HashMap<L, Assignment>>, Error>  
...
```

Type System

Advanced type level magic

```
pub fn solve_formula<F, C, L>(formula: F) ->
    Result<Option<HashMap<L, Assignment>>, Error>
where
    F: IntoIterator<Item = C>,
    C: IntoIterator<Item = L>,
    NonZeroI32: TryFrom<L>,
    Error: From<<NonZeroI32 as TryFrom<L>>::Error>,
    L: Abs<Result = L> + Eq + Hash,
{ ...
```

Further Arguments for Rust

Further Arguments for Rust

- Great ecosystem (tools, documentation, community)

Further Arguments for Rust

- Great ecosystem (tools, documentation, community)
- Easy verification (safety guarantees + type system)

Further Arguments for Rust

- Great ecosystem (tools, documentation, community)
- Easy verification (safety guarantees + type system)
- Strong academic connections (ownership type system, language semantics)

Further Arguments for Rust

- Great ecosystem (tools, documentation, community)
- Easy verification (safety guarantees + type system)
- Strong academic connections (ownership type system, language semantics)
- Multi-leveled native concurrency support (process vs. thread vs. async)

Why you might not want to use Rust

Why you might not want to use Rust

Ownership type system

- self-referential data structures
- complex mental model

Why you might not want to use Rust

Cumbersome rapid prototyping

- Rust enforces good design from start
- Workarounds are often verbose

Why you might not want to use Rust

Steep learning curve

- Borrowing and lifetimes
- unsafe
- Language quirks: e.g. orphan rule

Why you might not want to use Rust

Instability

- Language and compiler development
- No formal semantics (but active work)
- No specification (apart from Ferrocene)