**Comparison Report: Rust vs. C Implementation of the C4 Compiler**

**Overview**

The C4 compiler is a minimalist, self-hosting C compiler written in C by Robert Swierczek. Its Rust implementation was designed to maintain compatibility while taking advantage of Rust’s safety guarantees and modern tooling. This report compares the design impact, safety improvements, and development challenges between the two implementations.

**1. Impact of Rust’s Safety Features**

**Memory Safety**

* **C version**: Relies on manual memory management using raw pointers (char \*p, int \*e, etc.), with unchecked operations like malloc, memset, and direct pointer arithmetic.
* **Rust version**: Enforces memory safety through the use of Vec, Option, and safe ownership models. Raw pointers and unsafe blocks were minimized or eliminated.
* **Impact**: Prevented common C issues such as use-after-free, buffer overflows, and memory leaks.

**Lifetimes and Borrowing**

* Rust’s strict lifetime and borrowing rules required clear structuring of the compiler into components like Lexer, Parser, and VM, with controlled ownership and clear data flow.
* Global mutable state in C had to be refactored into scoped and encapsulated data in Rust, improving modularity and testability.

**2. Performance Considerations**

While no numerical benchmarks were collected, performance differences were observed in the following ways:

* **Startup & Execution**: The C version starts and executes slightly faster due to minimal abstraction and lack of safety checks.
* **Rust Overhead**: Rust’s safety features introduce minor overhead but offer predictable and stable behavior, especially when compiled in release mode.
* **Optimizations**: Rust’s LLVM backend often produces optimized machine code, meaning real-world performance is close to C for most programs the compiler handles.

**3. Challenges and Solutions in Maintaining C4 Compatibility**

|  |  |  |
| --- | --- | --- |
| **Challenge** | **Original C Behavior** | **Rust Adaptation** |
| **Pointer-heavy design** | Global pointers and memory regions manipulated directly | Replaced with owned types like Vec, slices, and references |
| **Global symbol table** | Flat memory layout with array indexing (id[Tk]) | Modeled as HashMap<String, Symbol> with strongly typed access |
| **Manual memory management** | Dynamic memory with malloc and pointer math | Used Rust’s allocation system (Box, Vec) and ensured safe memory reuse |
| **Expression parsing** | Recursive descent with little safety | Modeled with structured enums and clear control flow using match expressions |
| **Stack and VM operations** | Raw pointer-based virtual machine stack | Encapsulated stack in a safe Vec<i64>, with explicit bounds checks where needed |

**Conclusion**

Translating the C4 compiler into Rust preserved full functionality while improving code safety, structure, and maintainability. Rust's type system and memory model guided a cleaner and safer implementation, though it required refactoring away from low-level C idioms.

The Rust version trades off some raw performance and simplicity for correctness, clarity, and security — making it more suitable for long-term development and educational use.