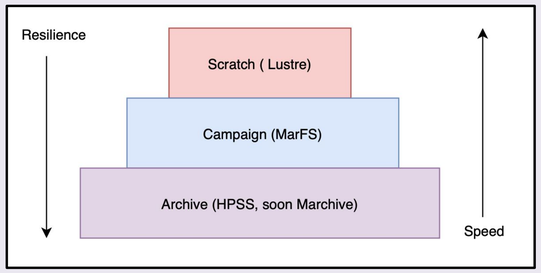
MarFS is Getting Rusty

Benjamin Schlueter, Clemson University (benjamin.schlueter1@gmail.com) – Mentors: Dave Bonnie, Garrett Ransom (HPC-INF)

Background

MarFS is LANL's in house campaign storage system, serving as a middle tier between scratch and the archive. It is designed to be as safe and redundant as possible, while providing adequate speed. There is interest in converting MarFS from C to Rust to further improve resilience, however, a full rewrite is unrealistic. MarFS must be converted modularly, where individual parts are rewritten and integrated with the C. This project tests if integrating Rust with MarFS C code is realistic to implement and performant.



Rust vs C

C:
- programmer allocates and frees memory
- few data validity checks
- very efficient and low overhead

Rust:
- memory allocated and freed behind the scenes
- enforces strict rules to prevent bugs
- very efficient: no garbage collector and advanced optimization
- provides convenient high level features to the programmer

Rust Prevents:

- Segmentation Faults
- Double Frees
- Buffer Overflows
- Dangling References
- Race Conditions

Does Not Prevent:

- Memory Leaks
- Deadlocks
- Any memory errors in unsafe blocks
- Logic Errors

Proof of Concept

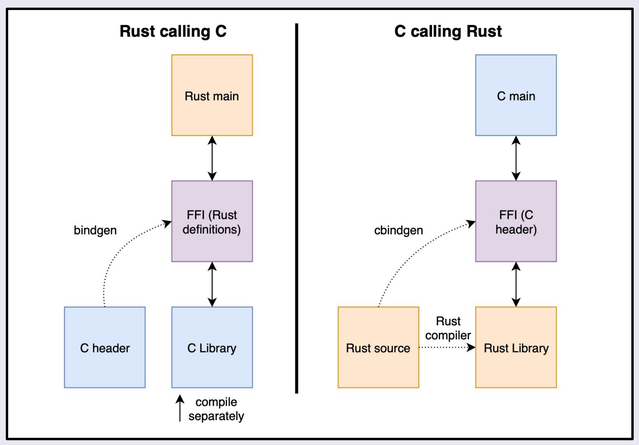
Two pieces of code were written to prove that Rust can be integrated with the MarFS C code. To do this, Rust's Foreign Function Interface (FFI) feature is used, which uses tools to convert code from one language into a form the other can understand. Once the FFI definitions are generated, the library with the implementations must be built / linked. The sections/figure below show the integration process.

`streamutil`

The first piece of code is the streamutil tool, which gives an admin a command line interface to call low level MarFS functions. This is a Rust application that calls MarFS C functions. To generate the FFI, the Rust provided `bindgen` tool was used to parse the C header into Rust definitions. Compiler variables are set to link the MarFS C library to the Rust code.

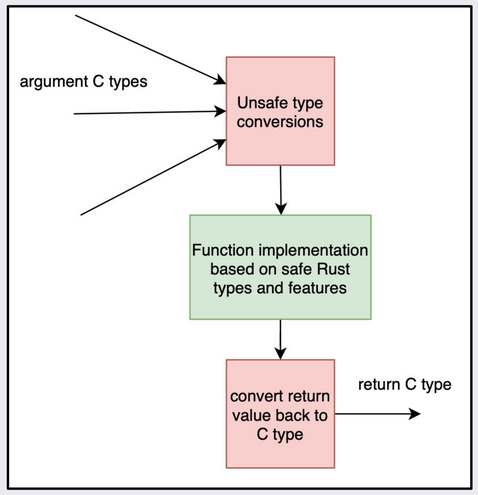
Data Abstraction Layer

Second is the Data Abstraction Layer, which is provides higher levels of C code with an object store like interface for data access. Similar to `bindgen`, the `cbindgen` tool parses a Rust source file into a C header. A variable can be set to tell the Rust compiler to output a static or shared library that C can link to.



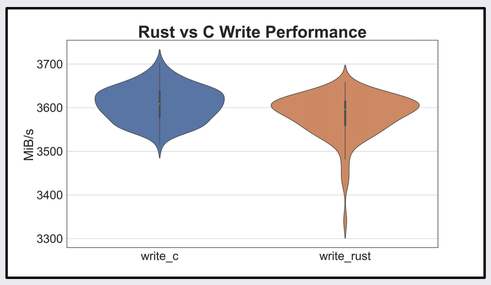
Implementation Details

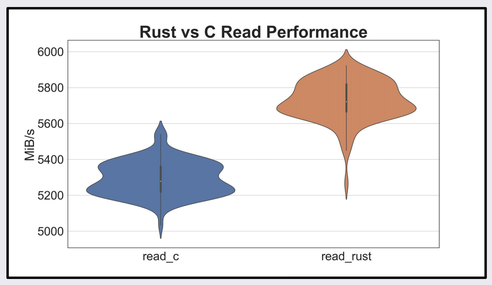
The integration process is easy to automate with a build script. Once automated, it should work hassle free. All types and features in the FFI must be C compatible. When calling a C function from Rust, unsafe type conversions from Rust to C types must be made. For example: casting `String` to `*mut c_char`. However, any Rust features can be used inside function definitions. The chart on the right shows the ideal structure of an C compatible Rust function. Also, advanced Rust features like Traits with no direct C translation cannot be in the FFI.



Performance

A benchmark was written to compare performances of the C and Rust DAL's. The benchmark involved a simple put of 200MB and a series of 1 MB gets at sequential offsets. Each plot contains samples from 5000 iterations. The data shown is from a virtual machine, where results were unpredictable. Often, the Rust would outperform the C, but because the VM was unstable, we are not drawing conclusions from these results. The benchmark was run on a real system and the Rust and C DAL's performed near identically as expected.





Conclusion

Integrating Rust and C is performant, simple, and easy to automate once the details are known. Rust as a language is pleasant to work with compared to C, with its efficient high level features and safety conventions. Smart Rust code is equally or potentially more efficient than C. Anyone who is looking to add some resilience and niceties to a program should consider Rust.

Managed by Triad National Security, LLC, for the U.S. Department of Energy's NNSA.
ROSY ID: 00fbd36a LA-UR-25-28194