Team 30

Hunter Belden, Pranav Nadimpalli, Elvis Palma, Ross Potter

Sponsor: General Dynamics Mission Systems
Mentor: Glen Uehara

Ira A. Fulton Schools of Engineering
Arizona State University

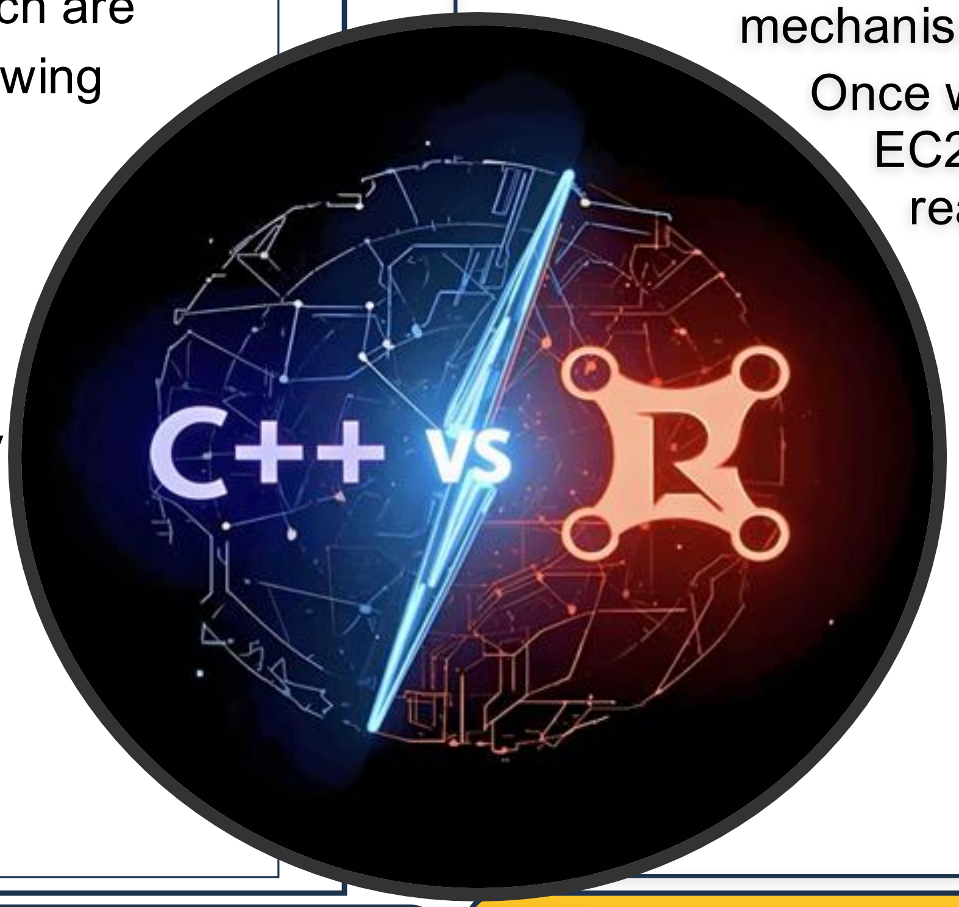
C++ V.S. Rust in Real-Time systems

Problem Statement

As embedded systems and real-time applications continue to grow in complexity and criticality, developers face increasing pressure to ensure both high performance and system reliability. Traditional real-time programming languages such as C and C++ offer low-level control and high execution speed but are prone to memory safety issues, concurrency errors, and undefined behavior that can compromise system stability.

Rust, a relatively new systems programming language, promises memory safety, thread safety, and high performance without relying on a garbage collector,features that directly address many of the challenges faced in real-time and embedded development. Rust's ownership model eliminates common bugs such as null pointer dereferencing, data races, and buffer overflows at compile time. This ensures safer memory management and more predictable performance, both of which are critical in electronics applications where timing precision and hardware control are essential. Additionally, Rust's growing ecosystem for embedded systems, including crates like embedded-hal and support for cross-compilation, make it increasingly practical for programming microcontrollers, Raspberry Pi boards, and other low-level hardware platforms.

Therefore, this project seeks to investigate whether Rust can serve as a viable alternative to C and C++ for real-time system development. By implementing identical real-time applications in Rust, C, and C++ on a Raspberry platform and measuring execution time, memory usage, and latency, the study aims to provide data-driven insight into Rust's performance, reliability, and potential advantages or trade-offs in real-time computing and electronic system design.



Research / Approach

Our research for this project focused on identifying and validating the specific advantages Rust Provides over C++. While both languages are widely used in systems programming. Rust's claims improved memory safety, concurrency guarantees, and development efficiency required deeper understanding. Rather than relying solely on anecdotal theoretical benefits, our team sought to understand these advantages in practical measurable terms that as our audience could easily interpret. To accomplish this we began studying the core language features that differentiate began studying the core language features that differentiate Rust from C++, including ownership and borrowing mechanisms, compiler-enforced safety rules, and modern tooling. Once we established these theoretical benefits, using AWS EC2 servers, we we designed experiments to quantify their real-world impact.

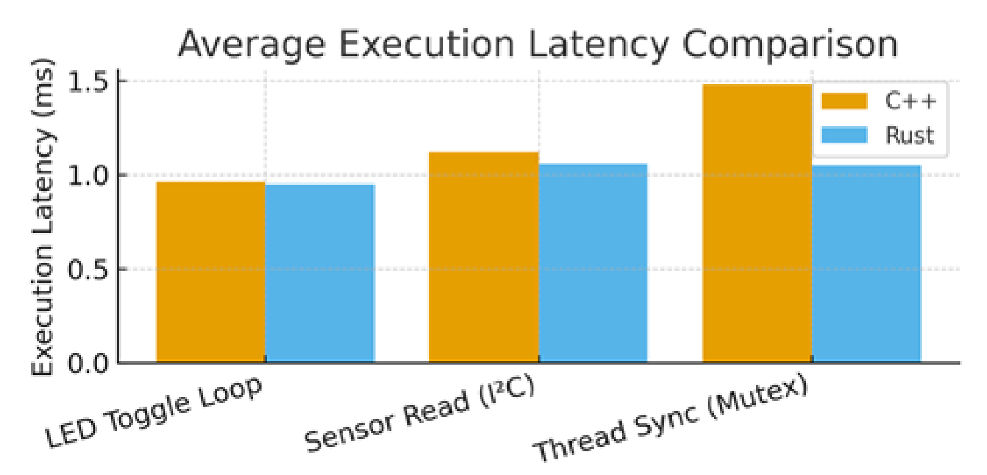
Identical Benchmarks: Implemented parallel C++ and Rust applications performing the same real-time I/O tasks on NXP LS1046A-RDB and Raspberry Pi 3B+.

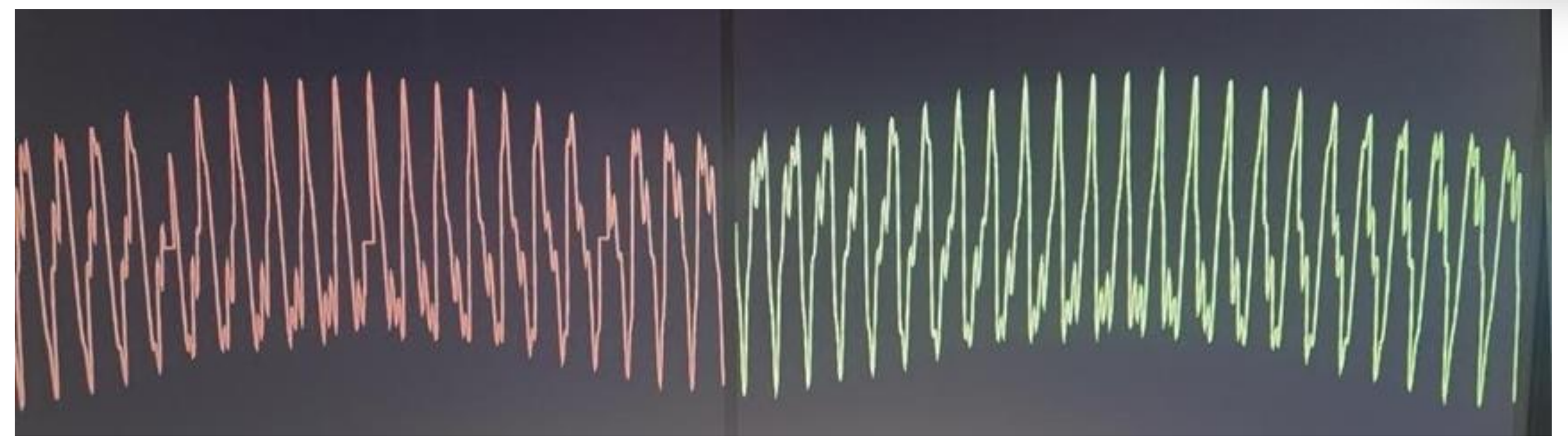
Metrics Measured:
-Execution latency (ns – ms)
-Jitter under multitasking
-Memory safety / fault resilience (via fuzzing)
-CPU & memory utilization

Data Capture:
Python heartbeat visualizer and log parser collect real-time timing data from both systems.

Analysis:
Python and Excel used for averaging, standard deviation, and graphical output of crash statistics.

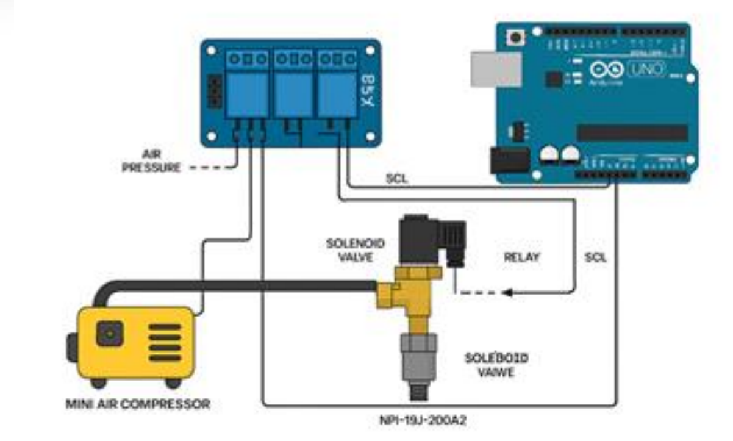
Data

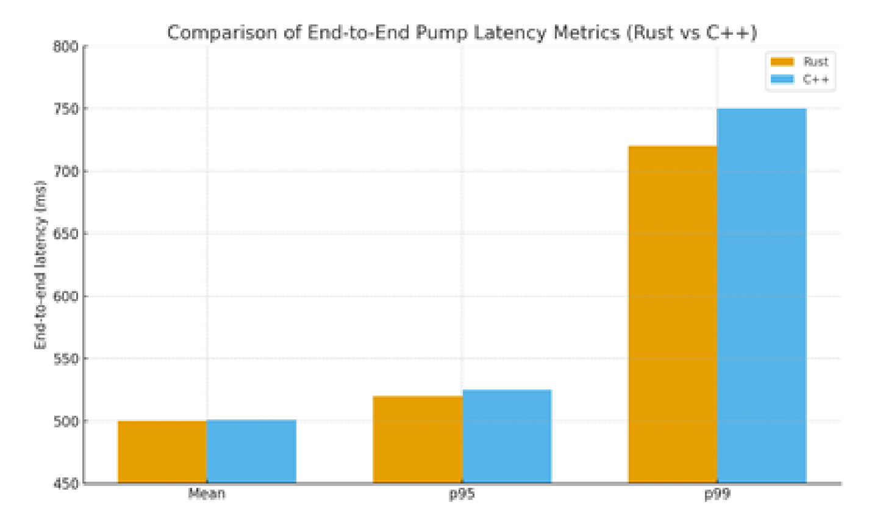


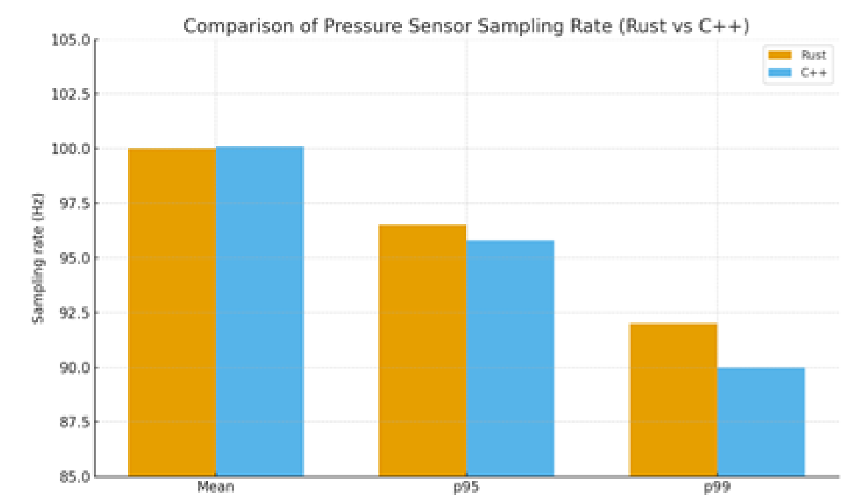


On the Raspberry Pi 3B+, Rust showed lower average latency, tighter timing behavior, and zero crashes under identical real-time tasks compared to C++, which matched Rust in raw speed but displayed more jitter and occasional segmentation faults. However, heavily optimized C++ builds (e.g., `-03`, `-march=native`, real-time threading libraries) could narrow these gaps. Overall, Rust delivered more predictable and fault-tolerant behavior on constrained embedded hardware, but further testing with optimized C++ configurations is needed for a definitive comparison.

Data







In our experiments we compared the performance of Rust and C++ on a Raspberry Pi that controlled a pump through a relay and read pressure data over I²C. Each program measured the end-to-end latency from activating the pump to receiving the first valid sensor reading, as well as the steady-state sampling rate of the pressure sensor. The results showed that both languages achieved nearly identical average performance, and the small differences observed in the p95 and p99 values are most likely due to hardware timing jitter, I²C bus delays, and the Linux scheduler rather than the languages themselves. Overall, Rust and C++ performed similarly in this real-time hardware setting.

Conclusion

Our experiments in virtualized environments demonstrated that Rust is fully capable of meeting real-time performance requirements while inherently eliminating many of the memory-related vulnerabilities commonly encountered in C++. Although Rust performed at speeds equal to or slightly slower than C++ in certain benchmarks, it's safety guarantees and predictability offer a strong tradeoff in high-reliability applications. By extending our tests across multiple hardware platforms, refining our analytical methods, and completing the driver and kernel-level integrations, the project is now positioned to deliver a comprehensive and compelling demonstration. The goal of this demonstration is to clearly showcase Rust's advanced capabilities, it's ability to manage memory safely without a garbage collector, it's strong concurrency model, and it's potential performance advantages in real-world scenarios. At the same time, we aim to present a balanced perspective: while C++ continues to benefit from it's extensive and mature library ecosystem, Rust's tooling and community-driven development are expanding at a rapid pace, reflecting the language's growing adoption and long-term viability. Throughout this project, our team has gained significant experience in function design, modularization, and top-down system development. These insights not only deepened our understanding of both languages but also strengthened our ability to evaluate technologies critically and design systems with safety, performance, and scalability in mind.