Interim Report

Project Title:	An Analysis Suite for Rust's Advantages in Linux
Name:	Fahad Khan
Student Id:	20454859
Email Address:	efyfk3@nottingham.ac.uk
Programme:	BSc Computer Science
Module:	COMP3003

Contents

1 Introduction

very quickly get out of the way the point and what will be achieved. add a quick paragraph on exactly how optimisation is difficult. how something on many machines, optimisation needs to be perfect. how developing memory-safe easily can really help.

1.1 Context and Motivation

Linux is an operating system that runs on billions of computers worldwide, with its underlying kernel worked on by thousands of individuals. Unlike its prominent market competitors, the ongoing development effort is open-source, allowing any individual to view and contribute to the source code. Adopters continue to further advantages such as hardware versatility, security and all-encompassing software utility [1]. For example, the kernel modularises processor-specific code, allowing it to be compiled and ported to any processor. This allows Linux to be run across a great variety of network architectures and embedded systems, which take advantage of its powerful system capability [2]. Linux's growing capital and market share have increased interest and contribution from large corporations, such as Google and Intel, becoming dependants [3]. Given the scale of development and sheer prevalence of Linux distributions worldwide, questions are naturally raised about the safety and maintainability of the kernel.

As demonstrated by the July 2024 CrowdStrike incident, where a small update with a memory issue to kernel-level software caused global Windows outages, without safety guarantees a pivotal error may result in catastrophic propagation across an operating system. Therefore, it may be important to re-evaluate the foundation on which Linux is built. The Linux kernel is written almost entirely in C, a programming language pervasive in operating system programming due to providing developers with low-level memory control and minimal runtime overhead. Optimisation is difficult yet incredibly rewarding, with precise manipulation of memory addresses enabling efficient data structure operations [4]. However, C's lack of modern safety features risks developer errors in memory management such as buffer overflows or dangling pointers, compromising system security and stability. This creates incentive to explore safer alternatives for the kernel while maintaining efficiency.

Rust now becomes the main contender for Linux development, a modern systems programming language designed to provide memory safety guarantees without sacrificing the low-level performance essential for kernel development [5]. Due to the continuing frustration with memory and concurrency issues within Linux, the value of preventing the various harms endemic to C grew into the "Rust for Linux" movement, culminating in the official addition of Rust kernel modules to the codebase in 2022 [6]. This ongoing effort has raised significant contention over the second language's practical efficiency and the prospective maintenance burden, forming the central focus of the project.

To carefully address these concerns, this project will produce a software suite to assess Rust's practical advantages within the Linux kernel. An automated framework will benchmark performance by executing a comparative suite that tests Rust modules provided by the user against other languages and variables.

1.2 Related Work

Extensive related work has focused on quantitative performance benchmarks between Rust and C, with findings largely indicating that Rust introduces performance overhead and does not eliminate all bug classes [7], [8]. could even mention the United States' Biden Administration 2024 report on using memory safe languages, "Future Software should be Memory-Safe"

1.2.1 Research Gap

There may be a research gap in these assessments of Rust's advantages, as purely quantifying overhead is insufficient to capture the full trade-off, and the project will address this by including evaluation of high-level abstractions and built-in safety guarantees.

1.3 Aims and Objectives

can be the same, but more indepth for the project (individual aspects more clear)

2 Technical Background

This section will establish the necessary technical background and definitions used for the software project, which will systemically measure Rust's advantages within Linux. To effectively quantify Rust's effectiveness in this context, it is essential to identify the specific characteristics of programming language that are advantageous for general operating system development. Performance benchmarks will assess how Rust handles these characteristics, with additional tests such as failure rates under adverse conditions. These evaluation metrics also must account for C being the incumbent language within the Kernel, which has significant practical consequences for Rust's adoption and continued integration within Linux. This requires assessment of interoperability, long-term maintainability and developing a deep understanding of its abilities with direct comparisons to C whenever possible.

2.1 Evaluating Languages for Kernel Programming

General Requirements

- introduce and explain in technical terms what a kernel needs from a programming language. tie this into non-deterministic latency and garbage collection and maybe even zero cost abstraction, ruling out most languages
- deterministic latency (real-time constraints)
- no garbage collection / manual control of allocation [9]
- zero-cost abstractions & predictable code generation
- low runtime overhead / minimal runtime footprint
- fine-grained memory layout control & aliasing model
- safe concurrency primitives or provable absence of data races
- interoperability with existing C ABI and kernel build system

current sources to fit in:

- making an enhanced kernel, highlights good/bad aspects [10]
- improving the linux kernel itself, involved languages [11]
- ruling out high level languages [12]

Benchmarking Methods

Evaluating a programming language in a kernel context is an inherently difficult problem in software measurability. Software is not inherently fit for empirical analysis as its form heavily relies on the expression of the programmer. This can introduce subjectivity to quality metrics, as is reported for future memory safety by United States Office of the National Cyber Director [13]. This challenge is magnified in an operating system kernel as intricate interactions with hardware and concurrent processes mean its behaviour is not entirely deterministic which, as the report also notes, "hinders the capacity to reliably and consistently measure" its true performance and security characteristics. The goal is to therefore use industry-accepted empirical metrics alongside carefully planned novel ones to satisfy the research gap.

- Introduce and explain technically, how languages are benchmarked performance-wise. explain why the technical limitations are there, insert from above paragraph

2.2 The C Language Family and Linux

- Once proper language requirements are added, write in certain features how it suites/violates them. also specific benchmarking once that's there

2.2.1 C’s Memory Management

2.2.2 Concurrency, and such Issues

2.2.3 C as the Standard

why C is the main base to compare Rust with, the best to benchmark speed, kernel panics etc with.

2.3 Rust for Linux

2.3.1 Rust’s Memory Management

2.3.2 Unsafe Blocks

2.3.3 Potential Extra Benchmarking Methods for Rust

-> explain how i intend to benchmark rust code with the subjective maintainability criteria

3 Project Description

3.1 Design

3.1.1 Benchmarking Approach

3.1.2 Optimisation Tools

3.2 Methodology

4 Progress

5 Bibliography

[1] E. Siever, *Linux in a Nutshell*. O'Reilly, 2005.

[2] D. Abbott, *Linux for embedded and real-time applications*. Newnes, an imprint of Elsevier, 2018.

[3] J. West and J. Dedrick, "Open source standardization: The rise of linux in the network era," *Knowledge, Technology & Policy*, vol. 14, pp. 88–112, 2001, doi: 10.1007/pl00022278.

[4] L. O. Andersen, "Program analysis and specialization for the C programming language," 1994.

[5] N. D. Matsakis and F. S. Klock, "The rust language," Association for Computing Machinery, Portland, Oregon, USA, 2014. doi: 10.1145/2663171.2663188.

[6] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, Santa Clara, CA: USENIX Association, Jul. 2024, pp. 425–443. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/li-hongyu

[7] S.-F. Chen and Y.-S. Wu, "Linux Kernel Module Development with Rust," 2022, doi: 10.1109/dsc54232.2022.9888822.

[8] F. Garber, "Rust in the Linux Kernel: Analyzing Rust Implementations of Device Drivers," *Tuwien.at*, 2025, doi: 10.347s.2025.127963.

[9] Y. Chang and A. Wellings, "Garbage Collection for Flexible Hard Real-Time Systems," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1063–1075, 2010, doi: 10.1109/TC.2010.13.

[10] Z. Shao, "Advanced Development of Certified OS Kernels," *Yale FLINT Group*, 2015, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.367.9646

[11] S. Saha, J. Lawall, and G. Muller, "An approach to improving the structure of error-handling code in the linux kernel," *acm*, pp. 41–50, 2011, doi: 10.1145/1967677.1967684.

[12] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a POSIX kernel in a high-level language." [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/cutler

[13] The White House Office of the National Cyber Director (ONCD), "Back to the Building Blocks: A Path Toward Secure and Measurable Software," 2024.

The number of words in this is 1079