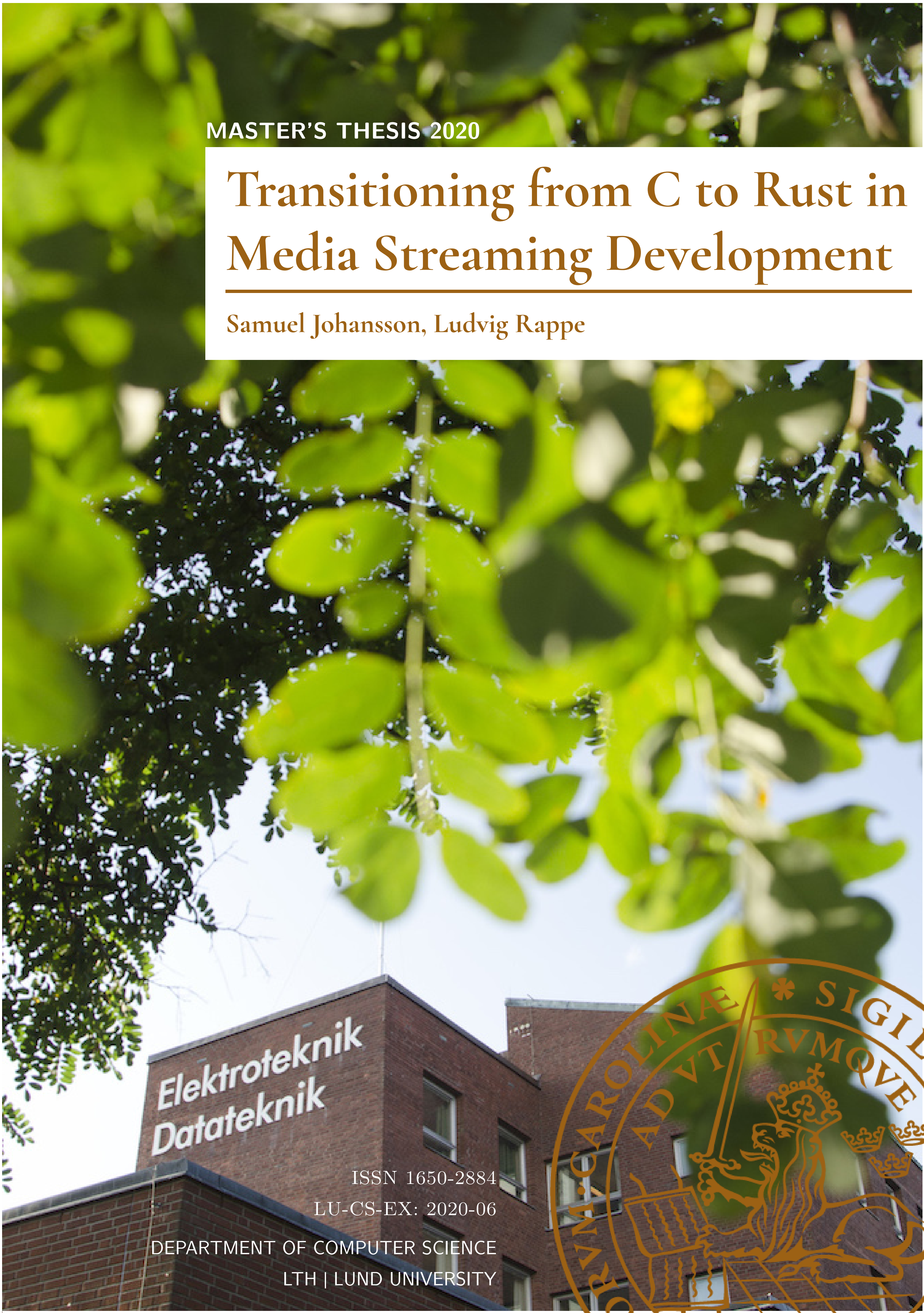

MASTER'S THESIS 2020

Transitioning from C to Rust in Media Streaming Development

Samuel Johansson, Ludvig Rappe

ISSN 1650-2884
LU-CS-EX: 2020-06

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-06

Transitioning from C to Rust in Media Streaming Development

Samuel Johansson, Ludvig Rappe

Transitioning from C to Rust in Media Streaming Development

(An Industrial Case Study)

Samuel Johansson
johansson.samuel@live.se

Ludvig Rappe
ludvig.rappe@gmail.com

February 25, 2020

Master's thesis work carried out at Axis Communications AB.

Supervisors:
Elizabeth Bjarnason, elizabeth.bjarnason@cs.lth.se
Jonathan Karlsson, jonathan.karlsson@axis.com
Srimanta Panda, srimanta.panda@axis.com

Examiner:
Björn Regnell, bjorn.regnell@cs.lth.se

Abstract

Rust uses static types, a strict compiler, and a unique ownership system, in order to make guarantees about memory safety without the use of a garbage collector and without sacrificing performance.

This paper presents a case study of evaluating a programming language transition from C to Rust through the creation and usage of a Programming Language Transition Framework.

We found the Programming Language Transition Framework to be a helpful tool for evaluating a programming language transition, but there is room for further improvements.

We found Rust to provide a far superior developer experience, while still performing on par with C. However, Rust's ownership system gives it a steep learning curve even for those with prior experience in programming.

Given that the impact on performance is acceptable, and that build system integration can be done successfully, we deem it feasible to transition from C to Rust in this case.

Keywords: Rust, C, Programming Language, Framework, Performance, Usability

Acknowledgements

First and foremost, we would like to thank Axis Communications AB for giving us the opportunity and resources to conduct this Master's Thesis.

We would also like to thank our supervisors at Axis, Jonathan Karlsson and Srimanta Panda. They have provided us with invaluable help and feedback, mostly regarding Axis-specific tools and the build environment, but also concerning the report and the Programming Language Transition Framework.

Last, but definitely not least, we would like to thank our supervisor at LTH, Elizabeth Bjarnason. She has helped us formalise the scope and methodology for this thesis and without her this thesis would not have been possible. In addition to this, she has also provided us with invaluable feedback regarding the report, how to conduct interviews, and much more.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
PLTF	Programming Language Transition Framework
IDE	Integrated Development Environment
OSS	Open Source Software
RAII	Resource Acquisition Is Initialisation

Chapter 1
Introduction

Today, there exist more programming languages than you can shake a stick at, and their number continues to rise. Choosing which languages to use can be a difficult task, as different languages offer different features. That task is also made more difficult due to the fact that one language is seldom strictly better than another. Instead, languages are often better in some areas and worse in others. After having used a certain programming language for some time, a new language could be released that may be better suited for your application. How do you determine whether or not you should transition to this new language?

In this chapter we present the context of the thesis, and briefly describe Axis Communications AB and its Streaming department. In addition, this chapter also gives a short introduction to the Rust programming language.

1.1 Context

The C programming language was created in 1972 [35], and is arguably one of the most widely used programming languages today. One of the reasons for this is that C provides good performance while also giving the developer a lot of control, e.g. by giving the developer direct control over their system's memory. However, this control comes at a price: discipline. C requires that the developer is disciplined enough to use the control given by the language in a responsible and safe way. If the developer does not have this discipline, then they might introduce a small error in the source code, which in turn could lead to severe errors and bugs in the program.

Rust is a programming language that was released in May 2015 (version 1.0), and aims to provide safety without sacrificing performance [24]. One feature that sets Rust apart from other languages is its *type system*, which builds on the accumulated experience and knowledge of C and C++ developers. This type system enforces many of the "best practices" that have been learned through years of programming, and enables the Rust compiler to make checks that find many errors (e.g. memory leaks) at compile-time.

1.2 Case description

Axis Communications AB (henceforth referred to as Axis) was founded in 1984 with a focus on developing networking technology for printers [4]. In 1996, Axis invented the world's first network camera, Neteye 200, and has since shifted its focus to primarily developing network cameras. Today, Axis continues to make advances in camera technology, while also successfully moving into other markets, such as access control and IP Video Doorstations.

The Streaming department at Axis is divided into multiple teams, each focusing on different parts of the development. The main goal for the department is to develop software for multimedia streaming between the host computer (referred to as *host*) and the camera (referred to as *target*). The department develops software for embedded systems (software to run on target), and for some of these programs the C programming language is used.

To achieve their goals, the Streaming department uses different tools and frameworks. Some examples of tools used are development environments (e.g. Vim [44], Emacs [12], and Eclipse[8]), tools for debugging (e.g. the GNU Project Debugger (GDB) [11] and Valgrind [43]), as well as tools for co-operation and code reviews.

One of the more important frameworks that the department uses is an *Open Source Media Framework*. This framework enables the department to create media pipelines by linking together various Media Framework Elements (referred to as *elements*). Thus, when the department wants to add new features, they can create a new element and insert it into an already existing pipeline.

The Open Source Media Framework used by the Streaming department is currently undergoing a gradual transition to Rust, while still supporting C. This is done by letting elements written in both C and Rust be used interchangeably in a pipeline. Since this framework has taken steps towards Rust, and because its creators are encouraging their users to do the same, the Streaming department is interested in learning the benefits and drawbacks of doing a transition from C to Rust.

1.3 Purpose

The purpose of this thesis is twofold: to find a structured way of evaluating a programming language transition, and to evaluate the feasibility of transitioning from C to Rust.

1.3.1 Research questions

First, we need to find a structured method that is suitable for evaluating a programming language transition in the given time-frame. We can then apply this method to investigate the consequences, both positive and negative, of doing a transition from C to Rust. In addition to this, we can also investigate what challenges one could face during such a transition. This is summarised by the following research questions.

RQ1 How can a transition from C to Rust be evaluated?

RQ2 What are the consequences of transitioning from C to Rust?

RQ3 What are the challenges of transitioning from C to Rust?

1.3.2 Approach

Our approach will be to first investigate what could be important to consider when transitioning from one programming language to another in general, and for Axis in particular. This information will then be used as a basis for further investigation into how the Streaming department at Axis could be affected by a transition from C to Rust. Based on our results, we will ultimately present our assessment of whether or not such a transition is feasible.

1.4 The four levels affected by a programming language transition

During this Master's thesis, we identified four levels that could be affected by a programming language transition. The four levels are Execution level, Programmer level, Toolchain level, and Management level. We refer to these four levels throughout the thesis, and they are explained in more detail in Chapter 4. However, in this section we will give a short introduction to them.

1.4.1 Execution level

The Execution level is the lowest of the four levels, the level closest to the hardware. This level can be seen as representing the interaction between the programming language and the hardware. Examples of factors covered by this level are execution time and memory usage.

1.4.2 Programmer level

The Programmer level covers the interaction between the programming language and the developer. This level focuses more on usability, i.e. what it is like to use the programming language as a developer. This level includes factors such as writeability and readability.

1.4.3 Toolchain level

The Toolchain level focuses on the interaction between the programming language and the tools. This includes tools for different purposes, e.g. debugging and development environments.

1.4.4 Management level

The Management level is the highest of the four levels, focusing on the "bigger picture". This can be seen as the interaction between the programming language and projects. This level covers how the programming language can affect the execution and management of projects, and includes factors such as productivity and code defects.

1.5 The Rust programming language

Rust is a fairly new programming language, but has since its release in May 2015 been voted the most loved programming language in the Stack Overflow Developer Survey every year [27] [28] [29] [30]. According to the Rust website [22], it is currently used in many popular applications, including Dropbox and the Mozilla Firefox web browser.

Using static types, a strict compiler, and a unique ownership system, Rust is able to make guarantees about memory safety without the use of a garbage collector. In practice, this means that many subtle memory and concurrency-related bugs can be detected at compile time, and that automatic memory management can be provided without sacrificing performance. Rust's high performance, combined with an emphasis on memory safety, makes it a suitable language for concurrent systems with strict security requirements.

In this section, we will give a short introduction to some of Rust's language features, and also briefly introduce Cargo [33], the Rust package manager.

1.5.1 Language features

Rust has some interesting language features that make it stand out from other programming languages. We will focus on covering these topics at a higher level, mostly describing the concept behind them and how they can affect the source code in general. As the focus is on a higher level, we will limit the amount of code examples we show in this section to only include the "Hello, World!" program shown in Figure 1.1. Readers interested in learning more about Rust are advised to read the book *The Rust Programming Language* [50] [37].

```
fn main() {
    println!("Hello, world!");
}
```

Figure 1.1: Hello, world! in Rust[37].

Ownership

One of Rust's more unique features is its Ownership system. This can be seen as a set of rules that dictate how data can be used. Rust's ownership rules state that each value has a unique owner, and that the value is dropped when its owner is no longer in scope. These can generally be good rules to follow in other languages as well, but unlike many other languages, Rust actively enforces them. Rust's Ownership system, Lifetimes (Section 1.5.1), and other language features, enable the compiler to make certain guarantees. One example of such a guarantee is that programs written in Rust will not have any memory leaks[1] nor data races (unless you use Unsafe Rust, described in Section 1.5.1).

[1]By using the function `std::mem::forget<T>()`, it is possible to leak memory in safe Rust. However, this is arguably not something a developer would accidentally do.

Lifetimes

Rust's automatic memory management is based on its lifetime system, which makes it possible to determine when memory should be freed. During compilation, the Rust compiler analyses references in the program to figure out their lifetime, which is defined by the Rust book as "the scope for which that reference is valid" [37]. The rules for determining the lifetime of a reference are called *lifetime elision rules*, and are patterns based on commonly occurring situations when writing Rust code.

For those situations where the rules are not enough to determine the lifetime of a reference, the programmer must annotate the lifetime manually. The lifetime elision rules are continuously expanded, which means that Rust's ability to automatically deduce lifetimes will improve as the language matures.

`Option<T>` and the absence of `null`

One special feature of Rust is the `Option<T>` enum. Values of this enum are either of some type `T`, or `None`, which indicates that there is no value. The value `null` was chosen not to be included in Rust, instead letting `Option<T>` with value `None` take that role. At first glance this may seem cumbersome, but this enables the compiler to check, at compile-time, if all the cases where `Option<T>` can be `None` are handled. If these cases are not handled, then the compiler will generate an error and inform the developer of this. Thus, using `Option<T>` instead of `null` enables the Rust compiler to detect errors that would not be detected until run-time in other languages.

`Result<R,E>` and error handling

The enum `Result<R,E>` is used for error handling in Rust. This enum is either of some type `T`, when we successfully return a value from an operation, or of some type `E`, when we return an error indicating what went wrong. Both of these cases must be handled when using a `Result<R,E>`, otherwise the compiler will generate a compilation error. `Result<R,E>` can therefore be used to get structured built-in error handling that is enforced at compile-time.

Object-Oriented Programming (OOP)

Rust also has some features that are inspired by object-oriented programming, with one of the most notable being the ability to have methods on structs and enums. Although Rust does not have the *object* keyword, Gamma et al. [47] defines objects as data and operations performed on that data. Following this definition, Rust could be considered an object-oriented language.

Another object-oriented feature in Rust is traits, which are similar to interfaces in Java. Traits make it possible to let many different types share behaviour through the methods defined on a trait. Even though functionality similar to object inheritance can be achieved by using traits, Rust does not have inheritance specifically. Thus, if a language must have inheritance in order to be considered object-oriented, Rust is not an object-oriented language.

Functional language features: Iterators and closures

Rust has two noteworthy features that are inspired by functional programming: iterators and closures. In this section we will give a short description of them.

In the Rust book [37], closures are defined as function-like constructs that can be stored in a variable. They are also described as anonymous functions that have the ability to capture variables from the scope they are defined in. These properties mean that closures can be used to reduce code duplication within a method, without having to pass a large number of arguments for every call to them. Closures are also useful for declaring short functions that will be passed as an argument to some method or function.

Iterators are defined by the Rust book as "a way of processing a series of elements" [37]. Iterators are responsible for handling the logic of iterating over elements, which means that the programmer does not need to write that code manually. One useful feature of iterators is that they have iterator adaptors, which are functions that return another iterator. This means that calls to iterator adaptors can be chained together, enabling the programmer to write code that does complex processing of a sequence of elements in a concise and potentially more readable way.

Unsafe Rust

Rust normally protects the programmer from making many memory-related errors by doing static analysis at compile-time. However, this static analysis is conservative and will therefore not accept some code that is actually correct. Thus, in order to provide a way for the programmer to write such code, *unsafe Rust* exists.

Unsafe Rust can be used in an unsafe block, and will enable the programmer to do certain things that are not allowed in safe Rust. These include:

- Implementing unsafe traits
- Calling unsafe functions and methods
- Accessing and modifying mutable static variables
- Dereferencing raw pointers

Mutex in Rust

Locking a `Mutex` in Rust is done by calling its `lock()` function. This function returns the protected resource wrapped in a type called `MutexGuard`, which is responsible for releasing the lock when it goes out of scope. The `MutexGuard` type implements the `Deref` trait, enabling Rust to automatically convert a `MutexGuard` reference to a reference to the protected resource. In practice, this means that one cannot access the protected data without first taking the `Mutex`, and one cannot forget to release the `Mutex`, as this is done automatically when it goes out of scope.

1.5.2 Cargo

Rust's package manager, Cargo, is closely integrated with the language and provides many useful features. These features include: handling dependencies, calling the compiler with the appropriate parameters, and introducing conventions. We will only cover these topics briefly, and refer readers interested in learning more to *The Cargo Book* [33].

Compilation and dependency handling

Cargo uses a special configuration file called `Cargo.toml` to handle a project's dependencies and compilation flags. By defining a project's dependencies in this file (e.g. as links to different repositories), Cargo is able to check these when building the project. Cargo will then download and install any dependencies that have not already been installed.

In `Cargo.toml`, you can also specify which flags should be used when compiling, e.g. what level of optimisation should be used for a release build, or how the program should handle panics (errors during run-time). Thus, when you build your Rust project using Cargo, you do not have to think about dependencies and compiler flags, as long as they are defined in `Cargo.toml`.

Extensions

The functionality of Cargo can be extended by installing extensions. Two such extensions are `rustfmt` [26] and `Clippy` [5], both of which aim to help the developer in writing better code.

`rustfmt` can be used to change the formatting of the source code according to specific rules. This tool checks whether the project follows the desired code style, and if it does not, `rustfmt` changes the code accordingly. `rustfmt` can be used from the command line by running `cargo fmt`.

`Clippy` is used to lint the source code, finding code that is either outright erroneous or could be written in a better way. Thus, it helps the developer write faster, clearer code, and potentially also reduces the number of errors. `Clippy` can be used from the command line by running `cargo clippy`.

1.6 Contributions

Both authors have been part of, and have contributed to, every activity involved in the case study. For most of the activities (e.g. programming, literature study) it was possible to split the work in such a way that both authors could do similar or identical tasks, while other activities (e.g. focus group) favoured more asymmetrical roles. As for the report, each chapter and section was written and/or proof-read by both authors.

Chapter 2
Related Work

In this chapter we present work related to our study. These works range from evaluation of Rust in particular, to evaluation of programming languages in general. Some of these works showcase metrics and methodology that can be used when evaluating programming languages, while others present what developers value in programming languages. The Rust-specific works provide additional valuable information about the language. For each of these works, a short summary is given.

Further below, the results from previous work are presented, organised by the four levels (Section 1.4). These levels are: Execution level (interaction between language and hardware), Programmer level (interaction between language and developer), Toolchain level (interaction between language and tools), and Management level (interaction between language and project).

2.1 Overview

Rikte [55] conducted a study, also at Axis, evaluating the Rust programming language by porting a Linux daemon from C to Rust. He described his experiences from the porting process, but also covered topics such as performance, building and compiling, debugging, IDEs, productivity, and learnability. He also gave an introduction to the Rust programming language and how it differs from C. Rikte's study covered several topics investigated in our study. However, while Rikte evaluated Rust based solely on his own experiences, we instead base our evaluation on a combination of our own experiences and the experiences of others.

Wilkens [58] conducted a study where he implemented one program in three different programming languages (C, Go[1], and Rust) and compared the solutions. The program implemented was a "shortest path calculation based on real world geographical data which is parallelized for shared memory concurrency". The languages were compared both in regards

[1]"Go is an open source programming language that makes it easy to build simple, reliable, and efficient software." [36]

to performance (Execution level), and in regards to productivity (Management level). Although this study is similar to ours, it was mostly based on the author's own experiences, while our study is based on both our own experiences as well as the experiences of others. Another difference is that Wilkens evaluated Rust in the domain of High Performance Computing (HPC), while we focus on the domain of media streaming.

Light [52] created a Unix-like operating system similar to Weenix[2] in Rust. In his paper, he compared Rust to C/C++ (in regards to system programming) and discussed the benefits and challenges of using Rust, both on the Execution level, and on the Programmer level. Light's study is similar to ours in that he compared Rust and C, and did so by not only investigating performance, but also investigating the benefits and challenges of working with Rust. However, Light did this in the domain of operating system programming, not media streaming, and based his evaluation only on his own experiences.

Meyerovich and Rabkin [53] investigated what factors are involved in programming language adoption. They did this by analysing in total around 800 000 large software projects hosted on either SourceForge or Ohloh. They also analysed the results of several surveys with between 1 000 and 13 000 participants, all of which were programmers. Meyerovich and Rabkin's work is similar to ours in that they investigated topics such as what developers deem important in programming languages. Their work differs in that they did this on a more general level, and did not focus on a specific domain or a specific type of programming language. We instead focus on the opinions of those working in media streaming in the embedded domain, and limit our study to the low-level languages C and Rust in particular.

Bhattacharya and Neamtiu [46] investigated how the choice of programming language affects software quality and developer productivity. They did this by doing statistical analysis on the code bases of four large Open Source applications written in C and C++: Blender, Firefox, MySQL, and VLC. Their work is similar to ours in that they investigated how the programming language affected software quality and developer productivity, and that they compared C to a language with higher-level features. However, their focus was on C and C++ in particular, while we do a comparison between C and Rust. Another difference is that we limit our investigation to the embedded domain, with focus on media streaming in particular, while Bhattacharya and Neamtiu investigated a broader range of software.

2.2 Impact on Execution level

Rikte found that the performance difference between C and Rust varied. When Rikte measured the C version and the Rust version of the Linux daemon, he found that Rust had a 14% slower execution time. Rikte noted that during the porting process he found a code defect in the C code that he fixed in the Rust version, which had the side effect of increasing the execution time of the Rust version in the worst case. Excluding the worst case from the performance measurements resulted in the Rust version being 9% faster than the C version. However, it is not clear whether these results were averaged or a single measurement, nor is the exact difference in milliseconds mentioned. Rikte also found that the Rust implementation used 7% (3MB) more memory and had a 36% (225kB) larger binary, when comparing the smallest executable for each language.

[2]Weenix[45] is an operating system written in C, and is used for teaching purposes at Brown University.

After running two performance tests on the C and Rust implementations of the operating system, Light found that Rust had, on average, worse performance than C. In the first test, the execution time of doing multi-threaded access on a contested resource with a varying number of threads was tested. Averaged over 10 runs, the Rust implementation took 3.040 times as long as the C implementation. In the second test, the execution time of doing the system call `waitpid` was tested. Averaged over 1 000 runs, the Rust implementation took 2.036 times as long as the C implementation. Light argued that this indicated that different functions will result in different amounts of slowdown, and that the slowdown was due to something other than the language's compiler. He argued that memory allocation was likely to be the reason, since the Rust implementation did this more frequently and at a higher cost than the C implementation.

When Wilkens measured the three implementations (written in C, Go, and Rust) of the shortest path application, he found that the Rust version had the shortest execution times. Compared to the C version, the execution times of the Rust version ranged from being 35% (197 minutes) faster at 2 cores, to 98% (62 minutes) faster at 24 cores. Another finding from the tests showed that the parallel speedup from increasing the number of threads was largest in the C version for lower thread counts, while Rust and Go had larger speedup at higher thread counts. It is not clear whether the results were averaged or came from single measurements.

2.3 Impact on Programmer level

Rikte found using Rust to be a "very pleasant experience". His conclusions were based on his personal experiences from a developer's perspective. He expressed his appreciation of the Rust standard library and listed several examples of Rust constructs that he found were easy to work with: enums, structs, traits, and tuples. The productivity was also found to be high when using Rust, which he attributed to aspects such as the ability to do fast prototyping and the possibility of utilising functional programming. Rikte also concluded that "Rust definitely aids programmers in writing correct programs." He found that Rust's ownership and borrowing system (Section 1.5.1) ensured that code re-factoring could be done without fear of pointer-related mistakes. However, Rikte pointed out that the learning curve of the borrowing system was a bit steep.

According to Rikte, the documentation of Rust's standard library had detailed explanations and was easy to navigate, and the compiler error messages were user-friendly. Another finding was that using Rust lead to a shortening of the source code, which he states was partly due to the possibility of replacing imperative style C code with functional style Rust code. Rikte also found code readability to increase as a result of this functional approach.

Light found that one of the benefits with Rust was that it had more high-level language constructs than C. These include the ability to attach methods to structs, as well as scope-based and custom destructors, which allowed Light to use Resource Acquisition Is Initialisation (RAII)[3] semantics. Using RAII semantics one can, among other things, ensure that one will never access synchronised data without locking its mutex or fail to release the mutex when one is finished with the protected data.

[3]"The basic idea [of RAII] is to represent a resource by a local object, so that the local object's destructor will release the resource."[57]

Another benefit of Rust that Light mentioned is its type, lifetime, and borrow checking system, which "help eliminate entire classes of errors from the code" and "allow one to express the meaning of code in a much richer way than is possible in C". Light also mentioned that Rust is a type-safe language and that the standard library includes two important wrapper types: `Result<T,E>` and `Option<T>`. These types are explained in Section 1.5.1.

One challenge that Light encountered with Rust, was that the language does not have structure inheritance (Section 1.5.1). This forced him to rethink his code structure, and also resulted in duplicated code.

Light also encountered a critical problem when using Rust in the domain of operating system development. The problem he encountered was that when a heap-memory allocation failed, Rust would call `abort()` (terminating the process) but other than that the programmer would not be aware that the allocation failed. Light's study was made before the release of Rust 1.0.0 and the current version of Rust (1.32.0) still calls `abort()` when a memory allocation fails, however it does display an error print informing the user[4].

When Bhattacharya and Neamtiu investigated the differences in internal quality between C and C++, they used the cyclomatic complexity and interface complexity metrics, and found that the C++ code had higher internal quality than the C code. They attributed this to C++ being more of a high-level language than C.

Meyerovich and Rabkin found that developers could have different feelings about languages with similar semantics, suggesting that language perception is affected by experience and training. The enjoyment of a language was found to be highly correlated to expressiveness, elegant code, and the ease of abstracting patterns in code.

They also found that ease and flexibility were considered more important than correctness, and that developers were unenthusiastic and even uneasy about static typing. Developers were more inclined to emphasise the readability and safety benefits of types, rather than the benefit of finding bugs.

2.4 Impact on Toolchain level

Rikte found that Cargo, the Rust package manager, provides tools for benchmarking, building, dependency management, and testing. He also commented that the tools are "excellent and closely integrated and bundled with Rust".

Included with the Rust installation are the two debugging tools `rust-gdb` and `rust-lldb`, respectively requiring GDB and LLDB in order to work. Rikte found, when using `rust-gdb`, that "setting breakpoints, stepping into, stepping over, stepping out etc. all worked as expected" and that "the Rust pretty printers are very good". Rikte tested changing variables during runtime, as well as instantiating Rust structs and adding them to data structures, and commented that "no problems were found, and the experience was very positive". Rikte also notes that "multi threaded debugging worked very well" and that "the tools feel mature enough for production use".

Rikte evaluated two open source Integrated Development Environments (IDEs) that had Rust plugins: Eclipse RustDT and IntelliJ Rust. He found that "IDEs may not yet be full featured" but "development activity in this department is high, and the experience is continuously improving".

[4]We tested this by writing a simple program that tried to allocate more memory than was available.

Wilkens found Rust to offer “excellent tool support for dependency management and other parts of the build process like testing”. He mentioned this as a potential reason for him having a lower development time when using Rust.

2.5 Impact on Management level

Rikte found it easy to use C libraries from Rust code, as it required little effort. However, dealing with C strings was found to be cumbersome, as it required the use of unsafe blocks and many error checks.

Wilkens measured productivity using two metrics, total time and Source Lines Of Code (SLOC), and argued that low values of these metrics correspond to high values for productivity. Using these two metrics, Wilkens concluded that working with Rust was more productive than working with C, since the Rust code took less time to develop and had a lower number of SLOC. He argued that the longer development time for the C code could be due to the “manual implementation of common data structures” and the “high amount of memory and type related errors encountered during the development". Wilkens also mentioned that the large number of SLOC for the C implementation could be due to “the dominant bracing style” (placing curly braces on new lines) but that the difference between the C implementation and the Rust implementation was “way too significant to be only attributed to the style”.

Bhattacharya and Neamtiu evaluated external quality of C and C++ code. They represented external quality as defect density and measured this by using two metrics: defect count divided by effective Lines of Code (eLOC), and defect count divided by the change in eLOC (ΔeLOC). Across all applications, the average defect density for C was more than 6 times higher than C++ for both metrics. Because of this, they could conclude that C++ code is less bug-prone than C code.

Meyerovich and Rabkin found that the factor with most impact on the adoption of a language was the existence of open source libraries, followed by already existing code and expertise.

Chapter 3
Research Method

In this chapter, we present how we conducted our research, and give some motivation for our choice of methodology.

3.1 Overview

Since the main goal of this thesis is to investigate the impact of transitioning from C to Rust specifically for Axis, we chose to follow a case study approach based on the methodology suggested by Runeson and Höst [56]. The case study consists of five main parts, which can be seen in the overview in Figure 3.1.

First, we did a literature study in order to get an understanding of the work that has already been done in the area of transitioning between – and evaluating – programming languages. Equipped with information from the literature study, we were able to compile a list of factors that we deemed relevant when transitioning to another programming language. Based on discussions with our supervisors at LTH and Axis, we created the first version of the Programming Language Transition Framework (PLTF).

Next, we interviewed employees working in the Streaming department at Axis. This was both to get feedback on the relevancy of the factors in the PLTF, but also to get their insights and opinions regarding the topics covered by them.

We then translated a media streaming element from C to Rust in order to get experience working with the language in the environment of the Streaming department. We noted down our experiences during this process and measured the performance of the translated media streaming element.

Lastly, a focus group was held with Axis employees in order to gain additional feedback on the PLTF.

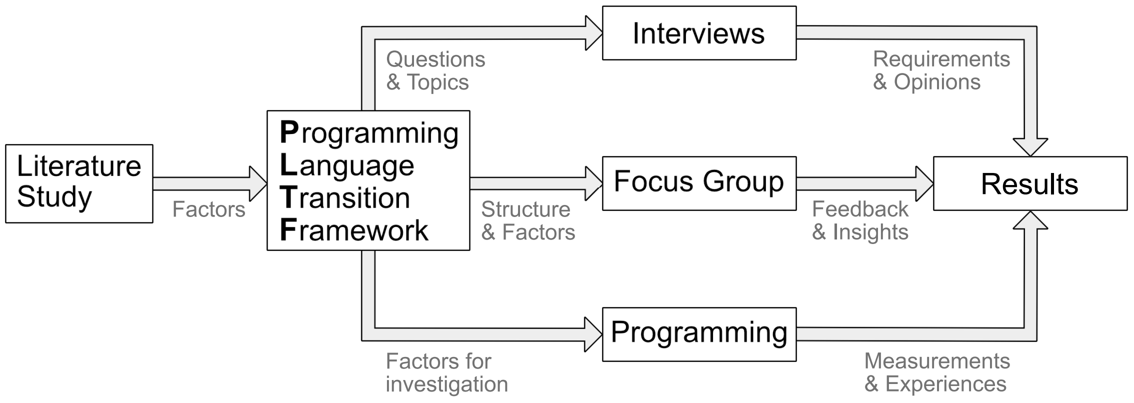

Figure 3.1: An overview of the case study.

3.2 Literature study

First of all, we wanted to see if similar studies had already been made and if so, what conclusions could be drawn from them. Second, we wanted to gather additional information that could help us find what factors are important when transitioning from one programming language to another. We did this by conducting a literature study, following a methodology inspired by Kitchenham and Charters [49].

We started the literature study by searching for related articles online (DuckDuckGo [7], Google [13], Google Scholar [14], IEEE Xplore [17], and LUBsearch [19]) using specific keywords. These keywords include, but are not limited to, "multiple programming languages", "c", "rust", and "programming language transition". We were also able to find articles comparing C and Rust through an older, online version of the *Rust Programming Language* [38].

Once we had found some relevant articles, and noticed that the keyword search produced diminishing results, we used backwards snowballing[1] in order to find more articles. When backwards snowballing started to produce diminishing results we decided to end that part of the literature study.

Using both the keyword search and backwards snowballing resulted in 55 articles that could be relevant to our study. We skimmed through all of these, and for each article we wrote a short summary and evaluated the article's relevance to our study. The more relevant articles were then read cover to cover.

3.3 Programming Language Transition Framework

As stated in Section 1.3.1, one of the goals of this thesis is to find a method for evaluating the feasibility of transitioning from one programming language to another. We also want this method to be a structured way of finding information about the effects of doing such a transition. This information can then be used to make a more informed decision about whether or not to do the transition.

[1]Looking at the references of the existing articles in order to find new articles that could be relevant.

One possible approach is to do a small-scale transition and evaluate it, but this would not be doable in the time-frame of our thesis. Thus, we started looking at more theoretical approaches instead. However, we were unable to find established methods for evaluating a programming language transition. After some discussions with our supervisors at LTH and Axis, we decided to create the Programming Language Transition Framework (PLTF).

The PLTF is structured as *a list of factors that could be affected by a programming language transition*. These factors are divided into four different levels: Execution, Programmer, Toolchain, and Management. For each factor, there is a description of what it represents, a motivation as to why it is included, as well as a suggestion on how to do the evaluation. The factors included in the PLTF were found through a process of brainstorming, literature study, and discussions with our supervisors at LTH and Axis.

After the literature study, we had a *brainstorming session* in order to come up with as many factors related to programming languages and their impact on software development as we could. Only the authors were present during this brainstorming session. We then refined these factors by merging those that were deemed to be closely related to each other, and grouped them based on our assessment of how they affect software development.

After further investigation, we found *additional literature* concerning the evaluation of programming languages, which we used to modify our list of factors. We also had *discussions* with our supervisors at LTH and Axis in order to get their feedback. This resulted in the version of the PLTF that was used when we created the interview guide (Appendix A).

Finally, we had a *focus group* that provided further feedback on the PLTF. This feedback can be found in Chapter 5.

3.4 Interviews

We conducted interviews with Axis employees working at the Streaming department. The primary purpose of these interviews was to *obtain information related to the factors* in the PLTF (see Chapter 4). A secondary purpose was to *get feedback on the PLTF* itself. The methodology we used is based on Runeson and Höst [56].

First of all, we created an *interview guide* (Appendix A) with prepared questions concerning the factors in the PLTF. These questions were created by both authors in a *brainstorming session*, and were later *reviewed by our supervisors*. The interview guide consists of ten main questions that are more open in nature, with each of these having closed and more specific sub-questions. The reasoning behind this structure is that we would first ask the main question, and then ask the sub-questions if they had not already been answered. Because of this, the interviews could be considered *semi-structured* based on the definition by Runeson and Höst [56].

We interviewed *eight employees* at the department, out of which two were managers and six were developers. We wanted to ensure as much variety as possible among the employees to be interviewed, and therefore selected them based on *what they worked with* and *how long they had worked at the department*. Some of the employees were also selected based on *suggestions from our supervisors*.

The interviews took 30-60 minutes, with seven of the interviews conducted in Swedish and one in English. During the interviews, one of the authors acted as the main interviewer while the other took notes and made sure that no questions were missed. These roles were the same during all eight interviews. All the interviews were *recorded* by both authors.

The interviews covered most of the interview guide, with some of the questions being excluded from the interviews with the managers, as those questions were not relevant in that case. Some sub-questions were not always asked as they were covered by other questions. An example of this is question Q2.4 “What language features do you like/dislike and how do they affect your work/productivity?”, which is covered by Q3 “What are your thoughts about C/C++?” (and its sub-questions) given that some developers mostly had experience with C.

After each interview we *transcribed the recordings*. One author would transcribe the interview based on their recording. Once that was finished, the other author would then read through the transcription, compare it with their recording, and correct any possible errors. Each author was the main transcriber for half of the interviews. The finished transcripts were then *sent to the corresponding interviewee* so that they could comment, correct errors, and elaborate on their answers if they felt it was needed.

Once the interviews had been transcribed, we *coded them using an immersion approach* [56]. First, we analysed and summarised the interview transcripts by writing down key points brought up by the interviewees. We then divided these into groups of identical or similar key points when possible. The results obtained through this process are presented in Chapter 5.

3.5 Programming in Rust

In order to evaluate Rust as a programming language, we needed to get some first hand experience using it. As neither one of us had any experience with programming in Rust, we first needed to learn the language. We did this by reading the book *The Rust Programming Language* [50] [37] cover to cover.

To be able to compare C and Rust, we needed a program written in both languages that could be used for comparison. Thus, we were provided with an *existing C program* from the department’s code base that we could *translate to Rust*. This program was selected by Axis based on its relevance to their work, with a size suitable for the time frame. The selected program is a *media streaming element* consisting of approximately 1000 lines of C code.

We started by *analysing the C code* and its tests in order to find out how the code worked and how it was used. Our initial plan was to first translate a test, and then translate the code that it was testing. However, we soon found that we could *use the tests written in C* to test the Rust code directly. There was therefore no longer any need to translate the tests, and we could instead *start translating the C code immediately*.

First, we created a *code skeleton* in Rust based on examples from the media streaming framework. This code skeleton contained the same functions as the C code, but without implementations. We then *ran the tests* in order to find out which functions failed first. These became a natural starting point for the translation. After we had translated the functions where the tests failed, we ran the tests again and found the next set of functions to translate. This process continued until all tests finally passed. Essentially, this meant that we could follow a *test-driven development* approach without having to write any tests.

During the translation process, we took notes on what we found good, bad, and challenging with Rust. As we had already created the PLTF at this stage, we had those factors in mind during this process. We later used what we had written down, combined with experiences that we could recollect from memory, to evaluate the usability of Rust.

Once the translation was complete, we ran tests to compare the performance between the C version and the Rust version. We ran three tests in total: two for measuring execution time, and one for measuring memory usage. These tests were also run on the C and Rust versions of a media framework element created by the developers of the media framework. The results from the tests, along with our usability evaluation of Rust, are presented in Chapter 5.

The first thing we tested was *execution time*. We did this by constructing two minimal media framework pipelines, one for each version of the element under test. These pipelines were then executed and timed by using the Unix `time` command [41]. The tests were run using 16 different input sets, 100 times each.

The second test was for measuring *latency*. This was done by using the same pipeline as in the execution time test, but with some modifications. Instead of measuring the run time of the entire pipeline, we measured the time from when the element received an input to when it produced the corresponding output. In this test, we measured the time using media framework timestamp functionality. These tests were run using 16 different input sets, 1000 times each.

The final test was of *maximum memory usage*. This test was done by executing the same pipeline that was used when testing execution time, but now the memory usage was monitored using the Unix command `top` [42]. The tests were run using 16 different input sets, 100 times each.

3.6 Focus Group

We held a focus group with Axis employees as a way to *obtain more feedback* on programming language transitions in general, and on the PLTF in particular. The methodology we used is based on Kontio et al. [51].

First of all, we prepared three questions to be used as basis for discussion on each of the four PLTF levels. We also prepared three questions concerning the employees' thoughts on the PLTF as a whole, programming language transitions in general, as well as what the first steps would be for the Streaming department when doing a transition. We attempted to formulate all of the questions in an open-ended fashion in order to not restrict the discussion too much. The questions for the focus group are presented in Appendix B.

In order for the focus group to be effective, we limited the number of participants to six. These six employees were selected by Axis based on their knowledge about the covered topics.

During the focus group, one of the authors was responsible for keeping the discussion alive by asking prepared questions, as well as follow-up questions where deemed appropriate. We often encouraged the employees to participate in the discussion by stressing that all opinions are valuable. In addition to this, we sometimes posed follow-up questions to a specific participant in order to get them involved.

The second author was responsible for advancing the slides, noting down what was said, and summarising the findings after the discussions on each level. The reason for this was to reduce the risk for misinterpretation and provide an opportunity for the participants to add anything to what had been said.

The focus group conducted held in Swedish, and had a duration of approximately one and a half hour. Immediately following the completion of the focus group, we summarised and wrote down the findings based on our notes and what we could recollect from memory. The results from the focus group are presented in Chapter 5.

Chapter 4

Programming Language Transition Framework (PLTF)

We designed a framework to be used as a tool to aid in evaluating a programming language transition, which we call Programming Language Transition Framework (PLTF). The PLTF consists of a number of factors related to software development (summarised in Table 4.1). In this chapter, we describe the factors, motivate why they were included, and give examples on how to evaluate them. This chapter also gives some suggestions on how the PLTF can be used.

4.1 Factors

A transition to a different programming language affects factors at different levels of software development. We have identified four levels that are affected, namely the Execution level, the Programmer level, the Toolchain level, and the Management level. A summary of these factors is shown in Table 4.1.

4.1.1 Execution level

In this section we present the factors that are related to the execution of software. This includes factors such as execution time, memory usage, and size of executable. Most of these factors are rather quantitative and can be evaluated by writing software with similar functionality in different programming languages and measuring the difference.

Execution level		
E1	Execution time	How the execution time is affected.
E2	Parallelism	How the speedup gained from parallelisation is affected.
E3	Compilation time	How the compilation time is affected.
E4	Memory usage	How the amount of memory used during execution is affected.
E5	Size of executable	How the size of the compiled program is affected.
E6	Compatibility	How suitable the language is for specific tasks and hardware.
Programmer level		
P1	Readability	How easy it is to understand code.
P2	Writeability	How easy it is to write code.
P3	Learnability	How easy it is to learn the language.
P4	Knowledge base	How much knowledge is available about the language.
Toolchain level		
T1	Testing	How the usage of testing tools is affected.
T2	Debugging	How the usage of debugging tools is affected.
T3	Co-operation	How the usage of co-operation tools is affected.
T4	Compilation	How the usage of compilers is affected.
T5	Development	How the usage of development tools (e.g. IDEs) is affected.
Management level		
M1	Integration	How easy it is to integrate software written in the language.
M2	OSS collaboration	How OSS collaboration is affected.
M3	Productivity	How productivity is affected.
M4	Code defects	How work related to code defects is affected.

Table 4.1: A summary of the factors in the PLTF.

Execution time

The *Execution time* factor covers the time it takes to execute a task, where the definition of a task can vary from case to case. A task could be the execution of the entire program, but it could also be a smaller segment of code, such as a function call.

While the best way to measure execution time may vary depending on the context, a good rule of thumb is to decide on a task and measure the time it takes for the program to complete it.

We chose to include this factor since there is often a desire or requirement for software to execute tasks quickly. This factor is related to the factors *Time behaviour* [18] and *Efficiency* [9], as mentioned in other works.

Parallelism

The *Parallelism* factor covers the speedup gained from making code run in parallel.

The parallel speedup can be measured by comparing the execution time of a program running on a single processor, with the execution time of the same program running on – and taking advantage of – multiple processors. As a reference, the theoretical speedup predicted by Amdahl's law [48] can be used.

We chose to include this factor since writing parallel code has become increasingly common. This is unsurprising, as it is "one of the most important methods of improving performance" according to Hennessy and Patterson [48]. This factor is related to the factors *Time behaviour* [18] and *Efficiency* [9], as mentioned in other works.

Compilation time

The *Compilation time* factor covers how long time it takes to compile code.

Compilation time can, as the name suggests, be evaluated by measuring the time it takes to compile a program.

We chose to include this factor since compilation time can have a big impact on software development. This is especially true for projects with a large code base, where compilation can take a significant amount of time. This factor is related to the factors *Time behaviour* [18] and *Efficiency* [9], as mentioned in other works.

Memory usage

The *Memory usage* factor covers the amount of memory that is used during execution.

There are multiple ways to measure a program's memory usage. One of the easiest ways to do this is to start the program and then use a monitoring tool, such as `top` [42]. The problem with this method is that `top` displays the reserved memory, and not the actual amount of memory that is used by the program. Another method that does not have this issue is to run the program with `Valgrind` [43], which will display its actual memory usage.

We chose to include this factor since the amount of available memory in hardware can be limited, which is often the case for embedded software. This factor is related to the factors *Resource utilization* [18] and *Efficiency* [9], as mentioned in other works.

Size of executable

The *Size of executable* factor covers the memory required to store compiled code on disk.

The size of the executable can be found by measuring the size of code that has been compiled with the desired level of optimisation, and potentially also reduced in size through the use of tools such as `strip` [32].

We chose to include this factor since the amount of available storage space can be limited in certain hardware. This is often true for embedded applications. This factor is related to the factors *Resource utilization* [18] and *Efficiency* [9], as mentioned in other works.

Compatibility

The *Compatibility* factor covers the extent to which code is suitable for specific tasks and hardware.

One way to evaluate compatibility is to go through the programming language's features and design decisions, and for each one decide whether or not it is acceptable in the given context. This can be done in regard to the task, the hardware, or both. An example of this could be a programming language that has a garbage collector, in which case it would be important to determine whether or not the performance impact of this feature is acceptable.

We chose to include this factor since certain language features may be a deal-breaker in a given context. This factor is related to the factors *Functional appropriateness* [18], *Installability* [18], and *Portability* [9], as mentioned in other works.

4.1.2 Programmer level

In this section we present factors that are related to the interaction between the programmer and the source code. This includes factors such as readability and learnability. Most of these factors are qualitative and thus require qualitative measurement techniques. Evaluating these factors will probably require understanding of what developers like and dislike with different programming languages but also what makes some source code easier to read and understand than other.

Readability

The *Readability* factor covers aspects related to how easy it is to read and understand code.

Readability is often subjective, and it can therefore be difficult to measure in a quantitative way. The metric *Internal quality* [46] could potentially be used, but it is worth noting that the purpose of this metric is primarily to measure the complexity of code structure, and not the programming language itself. One can, however, look at specific aspects related to readability. Examples of such aspects are *how much is handled automatically by the language*, *the need for writing a lot of boilerplate code*, and *which programming paradigms can be used.*

We chose to include this factor since reading code is a large part of software development. If it is easy to read and understand code, then it is possible to start writing code sooner. This factor is related to the factors *Analysability* [18], *Easy maintenance* [9], *Readability* [9], *Expressiveness* [9], *Abstraction* [9], and *Memorability* [54], as mentioned in other works.

Writeability

The *Writeability* factor covers aspects related to how easy it is to write correct code.

Writeability can often be subjective in terms of programming style, and thus requires qualitative measuring methods. However, one can investigate to which extent the language helps the developer to write correct code.

We chose to include this factor since writeability is perhaps the most impactful aspect of software development. This factor is related to the factors *Writeability* [9], *Expressiveness* [9], *Abstraction* [9], *Efficiency* [54], *Errors* [54], and *Satisfaction* [54], as mentioned in other works.

Learnability

The *Learnability* factor covers aspects related to how easy it is to learn to use a language.

Learnability is subjective, as it depends on how much previous experience in programming the student has. Because of this, it can be useful to look at how similar the language is to other commonly used languages in terms of syntax and semantics. Examples of specific aspects that may affect learnability are *how helpful error messages are*, and *the availability of learning material*.

We chose to include this factor since learnability can be very important during a programming language transition. This factor is related to the factors *Learnability* [18], *Low traning time (learnability)* [9], *Consistency* [9], *Learnability* [54], and *Errors* [54], as mentioned in other works.

Knowledge base

The *Knowledge base* factor covers the availability of information about the language.

The state of a language's knowledge base depends on a number of factors, such as its age, its popularity, and its community. Specific aspects to investigate are *literature*, *documentation*, and *online forums* such as Stack Overflow.

We chose to include this factor since a large part of programming consists of looking up information related to the language or the problem to be solved. This factor is related to the factors *Learnability* [18], *Low traning time (learnability)* [9], *Satisfaction* [54], and *Learnability* [54], as mentioned in other works.

4.1.3 Toolchain level

In this section we present factors related to tools, e.g. tools for testing, compilers, and IDEs. These factors can be seen as both quantitative (what tools are available) and qualitative (how good are the available tools). However, one can argue that the quantitative aspect is more important for a transition, since one would most likely want to be able to use the same tools as before.

Testing

The *Testing* factor covers what tools are available for writing software tests in the language.

This can be evaluated by investigating which testing tools are available for the language. In the case of a programming language transition, it may be of interest to investigate if any tools that are already being used can be used with the new language.

We chose to include this factor since testing software is crucial for ensuring correctness in software development. This factor is related to the factor *Rapid development* [9], as mentioned in another work.

Debugging

The *Debugging* factor covers what tools are available for debugging software in the language.

This can be evaluated by investigating which debugging tools are available for the language. In the case of a programming language transition, it may be of interest to investigate if any tools that are already being used can be used with the new language.

We chose to include this factor since debugging is often necessary in order to solve errors during software development. This factor is related to the factor *Rapid development* [9], as mentioned in another work.

Co-operation

The *Co-operation* factor covers what tools are available for co-operation.

This can be evaluated by investigating which co-operation tools are available for the language. In the case of a programming language transition, it may be of interest to investigate if any tools that are already being used can be used with the new language.

We chose to include this factor since effective ways of co-operating is important in software development, especially for large-scale projects. This factor is related to the factor *Rapid development* [9], as mentioned in another work.

Compilation

The *Compilation* factor covers what compilers and other build system tools are available for the language.

This can be evaluated by investigating which compilation-related tools are available for the language.

We chose to include this factor since changing build systems can be a costly and time-consuming endeavour. This factor is related to the factor *Rapid development* [9], as mentioned in another work.

Development

The *Development* factor covers what development tools have support for the language.

This can be evaluated by investigating which IDEs and other editors are compatible with the language. In the case of a programming language transition, it may be of interest to investigate if any tools that are already being used can be used with the new language.

We chose to include this factor since developers may prefer a specific IDE or editor, and forcing a change could therefore result in developer discontent and a decrease in productivity. This factor is related to the factor *Rapid development* [9], as mentioned in another work.

4.1.4 Management level

In this section we present the factors that are related to management and the execution of a project.

Integration

The *Integration* factor covers aspects related to integrating a new language with existing systems and an existing code base.

Two examples of integration-related aspects to investigate are the *possibility to use both languages in the same project*, and the *possibility to do a gradual transition*.

We chose to include this factor since it can be costly to rewrite existing code. This factor was also motivated by Axis wanting to ensure the possibility of doing a gradual transition.

OSS collaboration

The *OSS collaboration* factor covers aspects related to the interaction and collaboration with OSS communities.

In some cases, OSS collaboration related to software, tools, or frameworks can be affected by a programming language transition. If these have been developed for a specific programming language and are not supporting the new language, there may still be an interest to do so. By discussing this matter with those responsible, it is possible that they may agree to adapt to the new language on their own. In other cases, it may be necessary to contribute to this adaptation, which would require its own investigation.

We chose to include this factor since Open Source Software (OSS) can motivate the use of a language. Meyerovich and Rabkin found that the existence of Open Source libraries in particular had the most impact on programming language adoption among the factors they included in their survey (Section 2.5).

Productivity

The *Productivity* factor covers aspects related to how productive developers can be in the language.

Evaluating productivity is non-trivial as it is affected by other factors, such as *Readability* and *Writeability*, but also the *Toolchain factors*.

We chose to include this factor since it could affect a product's time to market, which is often an important aspect. This factor is related to the factors *Rapid development* [9] and *Easy maintenance* [9], as mentioned in other works.

Code defects

The *Code defects* factor covers aspects related to which code defects are present in the language, and how frequent they are.

This can be evaluated by investigating which types of code defects are possible in the language, and if any code defects are more common than others. Code defect density can also be measured by using the metric *External quality* (Section 2.5).

We chose to include this factor since it could affect the correctness, and the quality, of the software. This factor is related to the factors *Maturity* [18] and *Reliability and safety* [9], as mentioned in other works.

4.2 Usage

We found these factors to be useful for evaluating a programming language transition (Section 5.1.3, 5.2.3, 5.3.3, and 5.4.3), but they could also be relevant when evaluating a programming language in general. The following suggestions on how to evaluate a programming language using the PLTF is based on the approach we chose to follow, but there may be other methods to do this as well.

Depending on the PLTF level, different roles are affected, and they may therefore need to be involved in the evaluation of the factors. On the *Execution level*, the factors are mainly related to performance, and are the ones most likely to have concrete requirements. Because of this, quality assurance and developers are most likely to be affected. On the *Programmer level*, the factors cover the interaction between the developer and the language, which means that developers will be affected primarily. On the *Toolchain level*, the factors revolve around the tools that are used. This means that the people using the tools – most likely developers and testers – will be affected. On the *Management level*, the factors are more focused on projects as a whole, and may therefore involve many different roles. This will primarily affect managers and integrators, but potentially also developers and testers.

While it is possible to evaluate many of the factors in the PLTF without having any concrete requirements for the given context, it is often beneficial to be aware of them beforehand. The reason for this is that the specific aspects that need to be investigated may vary from case to case, and finding the requirements can make it more clear where the focus needs to be. Ways that can be used to obtain this information include, but are not limited to, *surveys*, *interviews*, and *focus groups*.

The method used when evaluating the factors varies depending on the PLTF level. On the *Execution level*, mainly quantitative performance measurements are involved. On the *Programmer level*, the factors are evaluated through qualitative measurements concerning developers' opinions. On the *Toolchain level*, the evaluation consists of investigating which tools are available for use with the language, as well as how suitable they are. On the *Management level*, evaluation method varies from factor to factor, and includes investigations, contact with collaborators, and analysis of other PLTF factors.

Chapter 5
Results

In this chapter we present the results we obtained from programming in Rust, interviewing Axis employees, and conducting a focus group with Axis employees. The methodology used for each of these activities can be found in Chapter 3. The results are first organised by level (i.e. Execution level, Programmer level, Toolchain level, and Management level) and then by method (i.e. Programming, Interviews, and Focus group). A summary of which factors were investigated using the different methods is shown in Table 5.1. These results are later discussed in Chapter 6.

5.1 Execution level

The results related to the execution level are focused mostly on execution time (E1). However, factors such as compilation time (E3) and compatibility (E6) are also covered. The results from programming in Rust consist mostly of our own quantitative measurements, while the interview results cover the more qualitative aspects.

5.1.1 Evaluation of performance

In this section, we present the results from our own performance evaluations of both the original and the translated media streaming element. We also present results from measurements of another media framework element that had already been implemented in both C and Rust.

ID	Name	Programming	Interviews	Focus group
E1	Execution time	X	X	X
E2	Parallelism		X	X
E3	Compilation time		X	X
E4	Memory usage	X	X	X
E5	Size of executable	X	X	X
E6	Compatibility		X	X
P1	Readability	X	X	X
P2	Writeability	X	X	X
P3	Learnability	X		X
P4	Knowledge base	X	X	X
T1	Testing		X	X
T2	Debugging	X	X	X
T3	Co-operation			X
T4	Compilation	X	X	X
T5	Integrated Development Environments		X	X
M1	Integration	X	X	X
M2	Open Source Software collaboration		X	X
M3	Productivity	X		X
M4	Code defects	X	X	X

Table 5.1: An overview of which methods yielded results for the different factors.

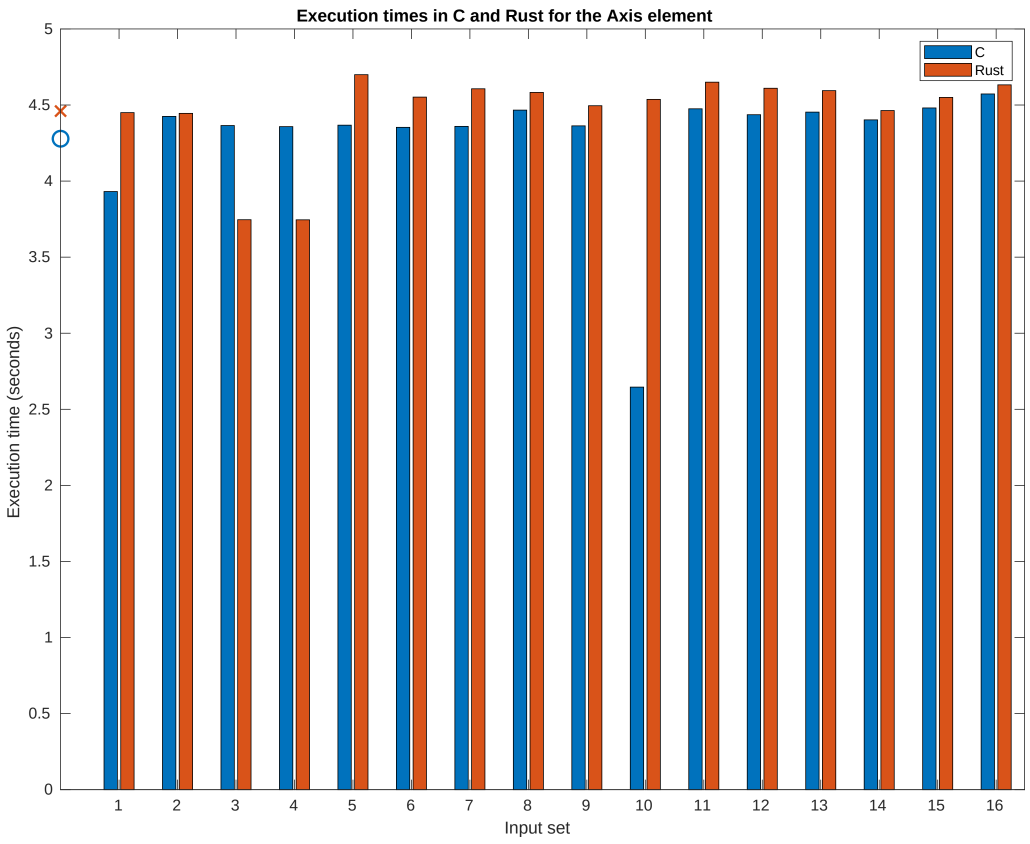

Figure 5.1: The execution times for the C and Rust version of the element, for different input sets. The execution times for each input set are the average of 100 runs. *O* and *X* mark the average execution time for C and Rust, respectively.

Execution time

In order to measure execution time, we ran one test on 16 different input sets. We did this 100 times each on both the C version and the Rust version (summarised in Figure 5.1). We found that the C version and the Rust version have similar performance, with the Rust version having an execution time that is, on average, 4.24% (0.18 seconds) longer than the C version. If we exclude data set 10 from these calculations (since it can be seen as an outlier), the average execution time for the Rust version is instead 1.54% (0.07 seconds) longer than the C version.

In addition to testing our own element, we also tested the C and Rust version of another element – written by the developers of the media streaming framework – using the same tests as before (16 different input sets, 100 runs each). The results from these measurements are shown in Figure 5.2. On average, the execution time for the Rust version of this element is 0.09% (0.004 seconds) longer than for the C version.

Another important metric is latency, i.e. the time from when the element receives an input to when the element produces the corresponding output. We measured this by running another test on the same 16 input sets, 1000 times each on both the C version and the Rust version. The results are summarised in Figure 5.3. We found that, in all tests, the Rust version has a greater latency than the C version. On average, the Rust version has a latency that is 25.69% (5.00 microseconds) greater than the C version.

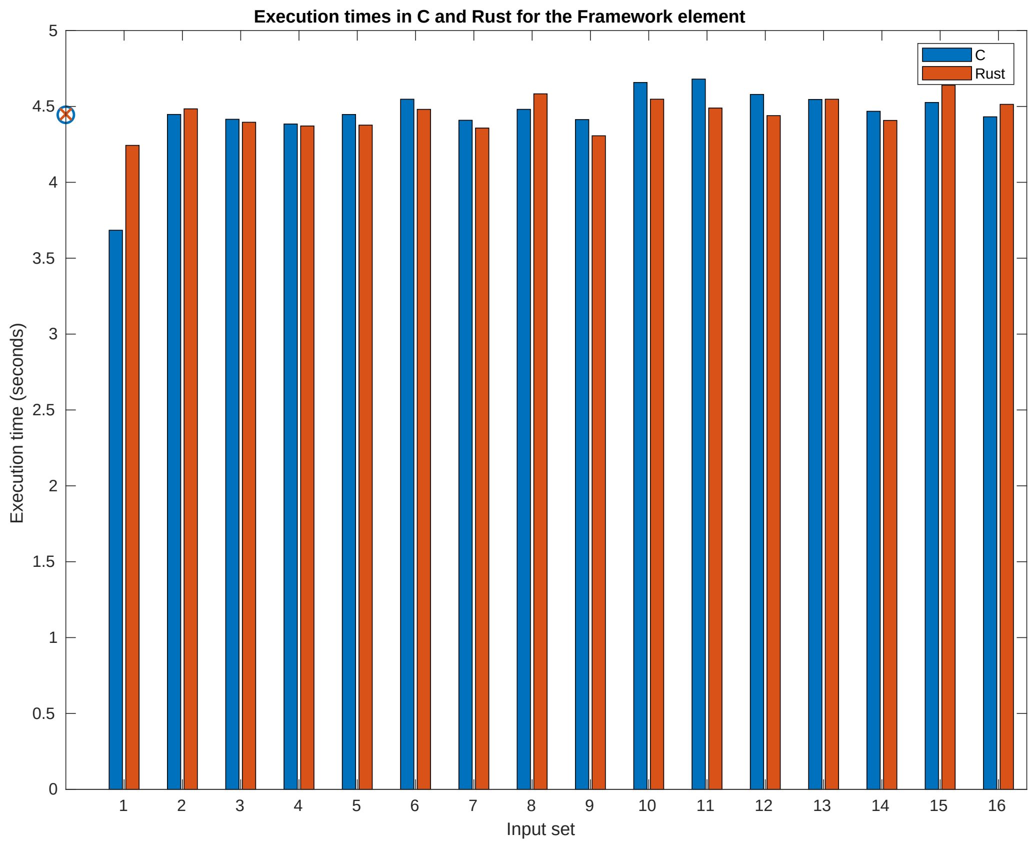

Figure 5.2: The execution times for the C and Rust version of an element written by the developers behind the framework, for different input sets. The execution times for each input set are the average of 100 runs. *O* and *X* mark the average execution time for C and Rust, respectively.

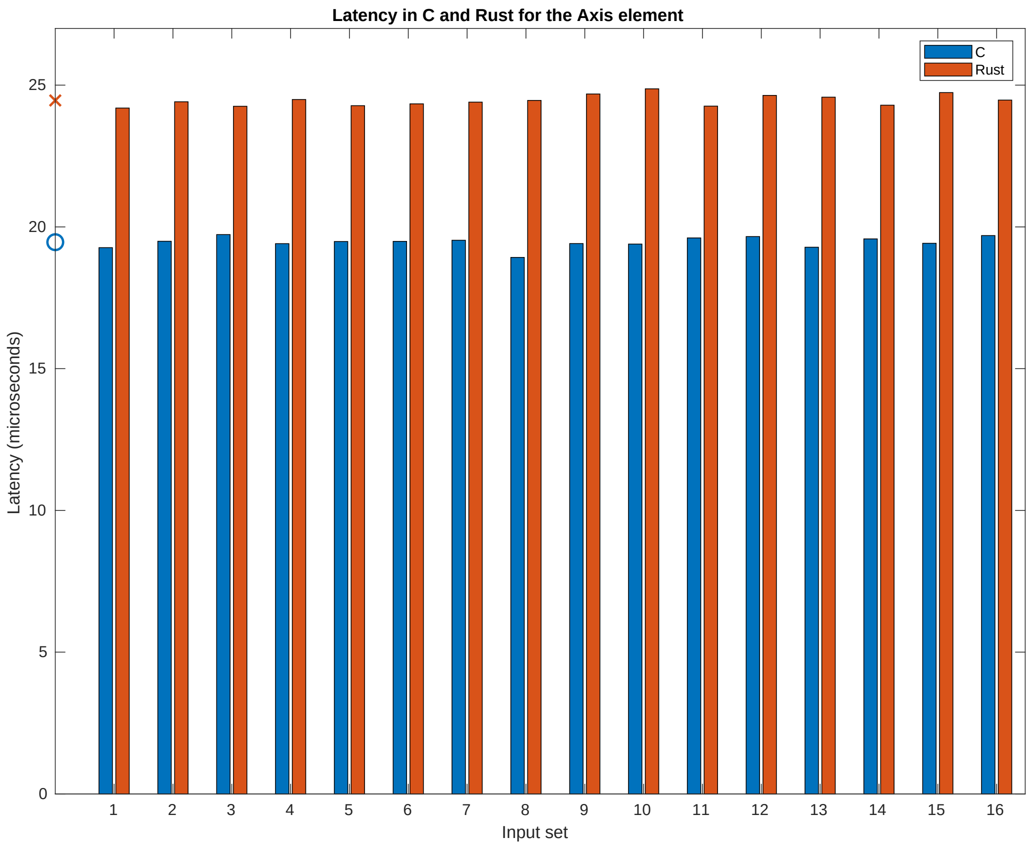

Figure 5.3: The latencies for the C and Rust version of the element for different input sets. The latencies for each input set are the average of 1000 runs. *O* and *X* mark the average latency for C and Rust, respectively.

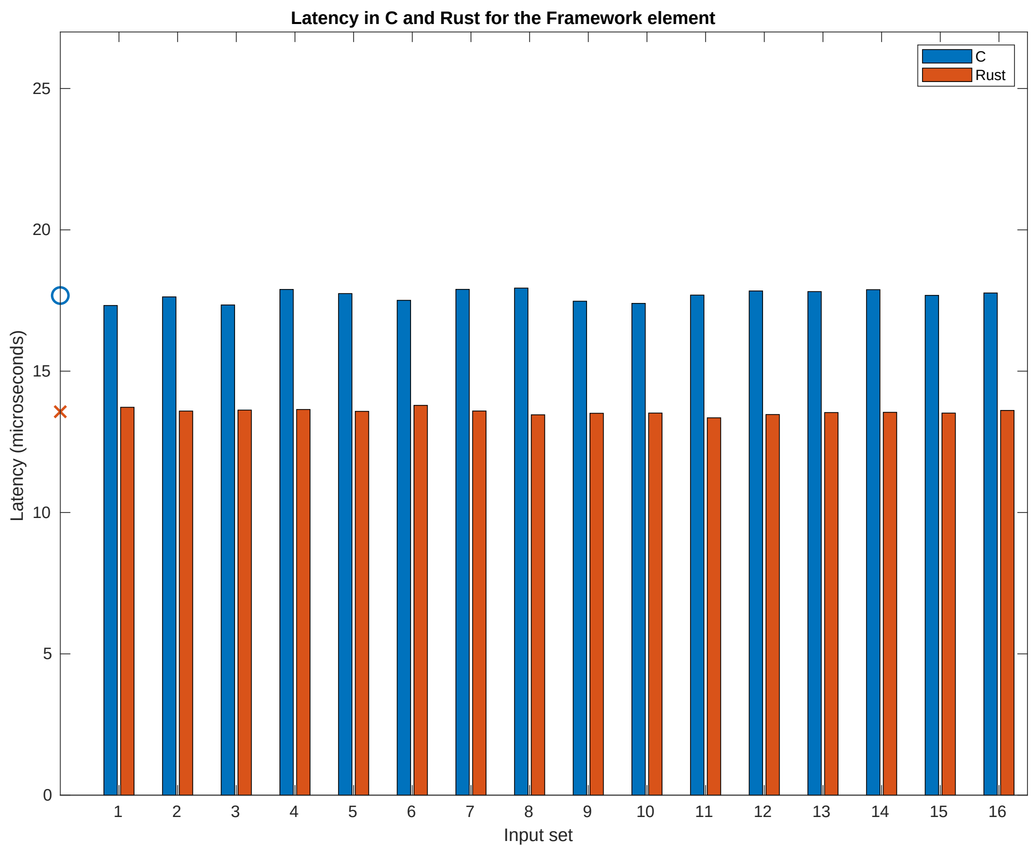

Figure 5.4: The latencies for the C and Rust version of an element written by the developers behind the framework, for different input sets. The execution times for each input set are the average of 1000 runs. *O* and *X* mark the average latency for C and Rust, respectively.

When we tested the latency of the element written by the developers of the framework (see Figure 5.4), we found that the C version's latency is 30.30% (4.11 microseconds) longer than that of the Rust version.

Memory usage

In order to measure memory usage, we ran a test on 16 different input sets, 100 times each, on both the C version and the Rust version. These were the same input sets that we used when measuring the execution time. The measurements were done on the entire pipeline, as it was not possible to do them for just the element. Figure 5.5 shows the results from these measurements. We found that, in all tests, the Rust version of the element uses more memory than the C version and, on average, the Rust version uses 1.656% (177 kbytes) more memory than the C version.

In addition to testing our element, we also tested an element written by the developers of the framework in both C and Rust (Figure 5.6). The C version of this element uses, on average, 43.71% (3129 kbytes) more memory than the Rust version.

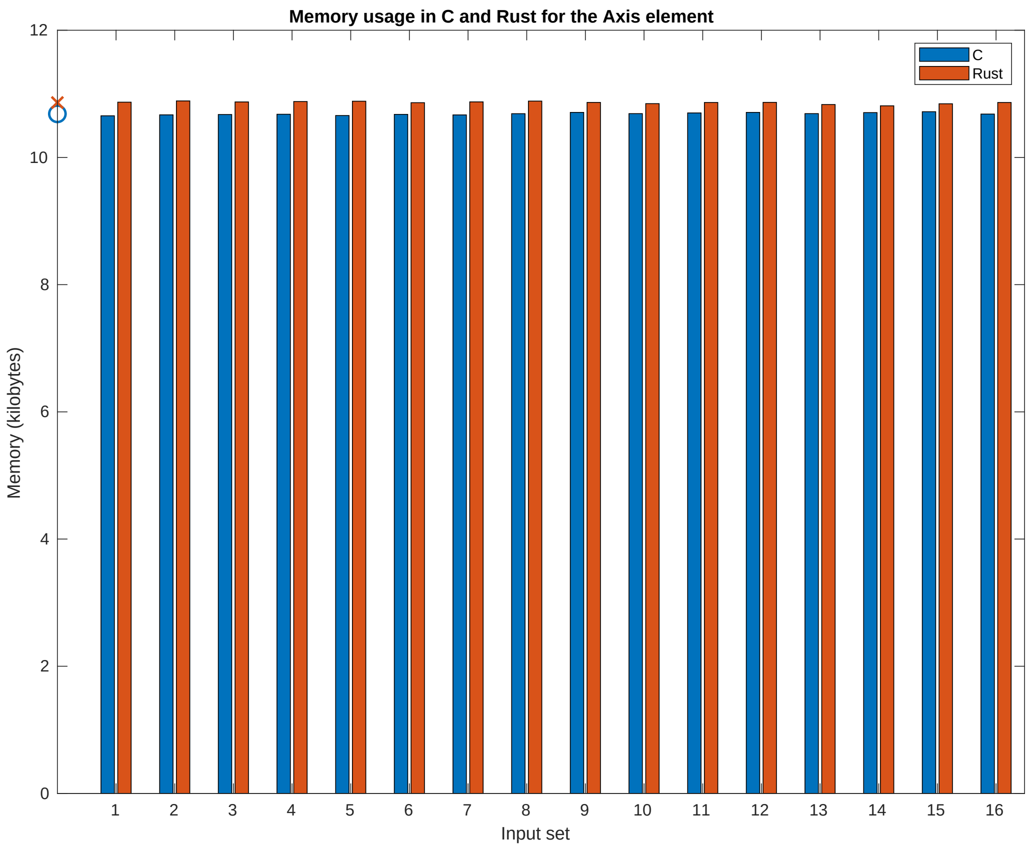

Figure 5.5: The maximum memory usage (maximum resident set size) for the C and Rust version of the element for different input sets. The memory usage for each input set is the average of 100 runs. The average maximum memory usage of each version has been rounded to the nearest integer. *O* and *X* mark the average memory usage for C and Rust, respectively.

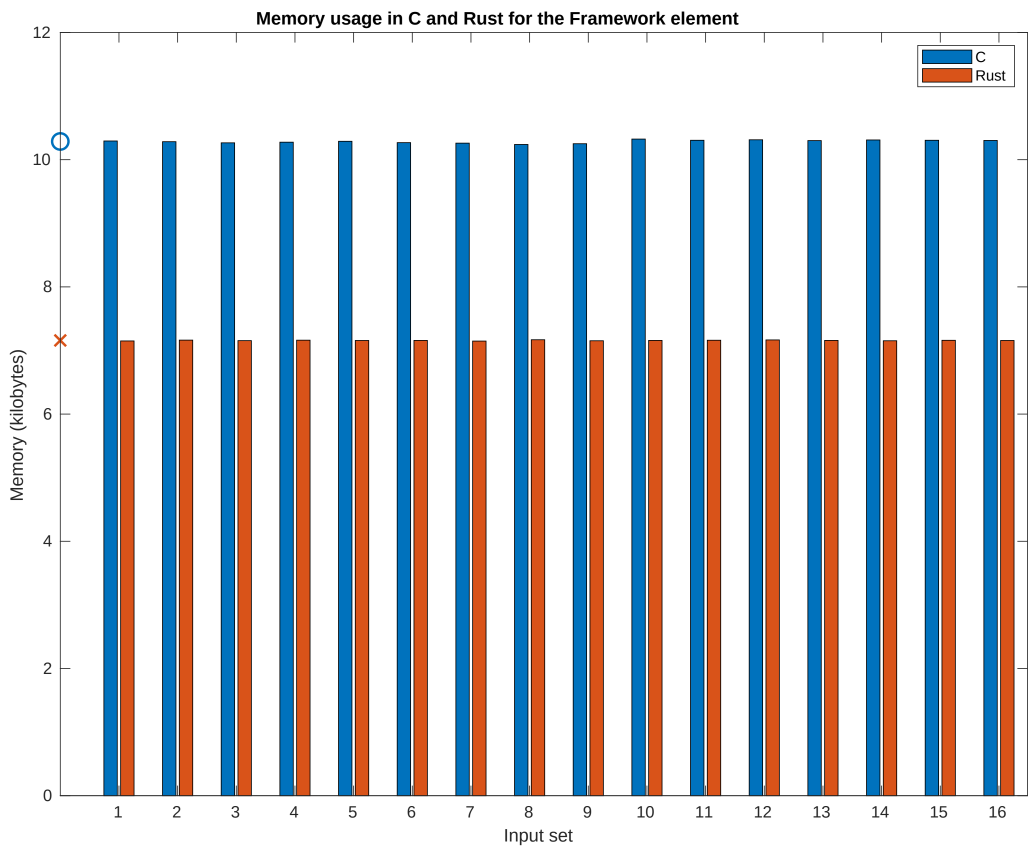

Figure 5.6: The maximum memory usage (maximum resident set size) for the C and Rust version of an element written by the developers of the framework, for different input sets. The memory usage for each input set is the average of 100 runs. The average maximum memory usage of each version has been rounded to the nearest integer. *O* and *X* mark the average memory usage for C and Rust, respectively.

Size of executable

We were not able to fully investigate the difference in size of executable between the C and Rust version of the element due to time restrictions. However, we did get some preliminary results that point towards Rust binaries being significantly larger than C binaries. We discuss this further in Chapter 6.

Compatibility

In embedded systems, compatibility with the hardware used in the different products is of course very important. When we tested our translated Rust version of the element, we had no hardware compatibility issues related to Rust. Unfortunately, since time did not allow testing on the target platform, we were only able to run the element on the host computer (64-bit Linux).

5.1.2 Performance requirements

During our interviews with the Axis employees, we asked them what they think is important for the Streaming department at the Execution level. In this section, we present what we learned from these interviews.

Execution time

When asked which performance aspects are important for their work, execution time was mentioned by most employees. One of the employees explained that they view execution time as something that is especially important to the customer.

A concrete performance aspect that affects the customer is that the specified frame rate should be upheld, which is directly linked to execution time. Since there are several different processes running on a camera simultaneously, it is important that all of them are efficient. A small increase in the CPU usage of one process may not seem to have that big of an impact by itself, but it will reduce the available CPU time for other processes, potentially affecting the overall performance significantly.

Another employee gave as an example that using a garbage collector, while helpful in providing automatic memory management, would be problematic in embedded software due to the performance impact that occurs when it starts cleaning up memory.

Parallelism

Since the processes running on the cameras execute in parallel, it is important that the language can handle concurrency effectively. However, when asked which performance aspects are most important, only one of the employees mentioned concurrency.

Compilation time

Compilation time was not considered to be an important factor by most employees, for two major reasons. First, compilation time affects neither the end product, nor the customer. Second, compilation time is a relatively small part of the total time used by the build system. This means that a change in compilation time will not have a large impact on the total build time.

Memory usage

An important performance aspect is memory usage, which was mentioned explicitly by all but one of the employees. One employee explained that there must not be too much overhead when allocating and deallocating memory, since a lot of memory needs to be allocated and deallocated for video frames all the time. The same employee gave their insight into why C is used in media streaming: "In a way, it's the great amount of data that is constantly handled that steers us towards using C."

Size of executable

Size of the executable is a performance aspect that was regarded as important by several employees. New features are constantly added, which requires the size of the programs to be kept small in order for everything to fit within the memory that is available on the cameras.

Compatibility

It was clear from the interviews that there are several hardware-related requirements that need to be met by the software before it can be included in the cameras, some of which have been explained earlier in this section. One additional important requirement is that the processor architectures in the cameras are supported by the language, which was something touched upon by one of the employees.

5.1.3 Focus group insights

During the focus group, it was concluded that all Execution level factors are important. It was also mentioned that it is difficult to rank them, due to the fact that priorities can vary depending on the product.

It was, however, clear that some factors are more important than others. For instance, the participants agreed that it is *important that all hardware is supported*, indicating that Compatibility is among the most important factors. We also learned that Execution time, Memory usage, and Size of executable are *limiting factors in the camera*, suggesting that these may be the most important Execution level factors in products with limited hardware. Parallelism was also regarded as important, but the developers were more interested in *how easy it is to write parallel code* in the language. It was also concluded that memory robustness is important, leading to the suggestion of a new factor: *Memory management*.

Compilation time was considered to be the least important factor, since it does not affect the end product directly, and it was suggested that this factor may fit better on the Toolchain level.

According to the focus group participants, it is *difficult to set specific requirements* on the Execution level factors. One exception to this is framerate, a metric related to Execution time, where there is often a concrete requirement that must be fulfilled.

It was also mentioned that *programming language culture can affect the execution level*. One example that was mentioned, was the extent to which libraries are used, which can vary between different programming languages. Using a library can reduce development time, but it may also increase the size of the executable.

5.2 Programmer level

The results related to the programmer level are focused on the interaction between the programming language and the programmer. The results from programming in Rust cover our own experience of using Rust. This includes what we found to be good, bad, or challenging when using the language. The interviews cover what the developers think of different programming languages, what they think makes code easier to read, what language features they like, and why they like some languages more than others.

5.2.1 Evaluation of programming in Rust

During the translation of the media framework element, we took notes of both the good and the bad experiences we had with Rust. In this section, we present our thoughts on programming in Rust based on these experiences.

Readability

We found that not having to handle certain things manually in Rust made the code more readable. We define such manual handling of things that could otherwise be handled by the language as *boilerplate code*. Our experience is that such code can be distracting when trying to understand the code logic.

An example of Rust reducing boilerplate code is its automatic memory management. Rust follows a pattern called Resource Acquisition Is Initialisation (RAII) (also known as Scope-Based Resource Management), which means that when a variable goes out of scope, its associated memory will be returned [37]. As a result, there will be less memory handling code (such as calls to `malloc` and `free` in C), that can be distracting when trying to understand what the code does.

Another example of boilerplate code reduction is Rust's implicit deref coercion, which is used to automatically convert from a reference of a type implementing the `Deref` trait, to a reference of the target type specified in the trait [37]. While this does remove certain manual conversions, generally making the code *easier to read*, it can also make it harder to *understand what the code does*.

We realised that we could take advantage of implicit deref coercion to avoid doing some conversions manually, and did so without any issues until we encountered a specific situation similar to the one in Figure 5.7. Here, we wanted to get mutable references to two different fields of a struct that was protected by a `Mutex` (Section 1.5.1). When attempting to compile this code, the compiler produces an error message that can be seen in Figure 5.8.

```
use std::sync::Mutex;

struct MyStruct {
    x: i32,
    y: i32,
}

fn main() {
    let mutex = Mutex::new(MyStruct { x: 1, y: 2 });

    let mut mutex_guard = mutex.lock().unwrap();
    let x_ref = &mut mutex_guard.x;
    let y_ref = &mut mutex_guard.y;

    *x_ref += 1;
    *y_ref += 1;
}
```

Figure 5.7: Attempting to mutably borrow two fields from a struct that is protected by a mutex. This code does not compile (see error message in Figure 5.8).

The error message points out that the `MutexGuard` cannot be borrowed mutably more than once, which seems to be irrelevant in this case, as we are only interested in borrowing the two fields (mutably borrowing two fields from the same struct is allowed in Rust). What happens in the background when we try to borrow `x` is that `deref_mut()` is called on the `MutexGuard` in order to convert it into a `MyStruct` so that we can get a reference to the variable. In order to do this, the `MutexGuard` is mutably borrowed and stored at an unnamed memory location for as long as `x_ref` is in scope. When we then attempt to borrow `y`, the same procedure is followed, giving us another mutable reference to the `MutexGuard`. Even though this already seems to violate the ownership rules (only one mutable borrow of a piece of data is allowed), it actually does not as of Rust 1.31 thanks to the introduction of non-lexical lifetimes [25]. However, as soon as we try to use `x_ref`, the compiler sees that both references to the `MutexGuard` will have to be in scope at the same time, which is a violation of the ownership rules. The compiler therefore produces the error message shown in Figure 5.8.

The solution we found is presented in Figure 5.9, where we create one reference to the struct from the `MutexGuard` by calling `deref_mut()` manually. The reason this had to be fixed by borrowing the struct is that having only one mutable reference to the `MutexGuard` is not enough. In that situation, the struct would be borrowed mutably twice instead, since the two calls to `deref_mut()` return a separate mutable reference each. This shows that implicit deref coercion, while helpful in general, can negatively impact readability and writeability in that it is not always clear what is happening in the background from just reading the code.

What we found most helpful in terms of readability in Rust are the object-oriented programming features (Section 1.5.1). How much an object-oriented structure benefits readability, we consider to depend both on how well the problem fits this structure and on how well it has been implemented.

```
error[E0499]: cannot borrow ‘mutex_guard‘ as mutable more than once at a time
  --> src/main.rs:15:22
   |
14 |     let x_ref = &mut mutex_guard.x;
   |                      ----------- first mutable borrow occurs here
15 |     let y_ref = &mut mutex_guard.y;
   |                      ^^^^^^^^^^^ second mutable borrow occurs here
16 |
17 |     *x_ref += 1;
   |     ----------- first borrow later used here

error: aborting due to previous error

For more information about this error, try ‘rustc --explain E0499‘.
error: Could not compile ‘mutex-problem‘.

To learn more, run the command again with --verbose.
```

Figure 5.8: The error message that is produced when attempting to compile the code in Figure 5.7.

```
use std::sync::Mutex;
use std::ops::DerefMut;

struct MyStruct {
    x: i32,
    y: i32,
}

fn main() {
    let mutex = Mutex::new(MyStruct { x: 1, y: 2 });

    let mut mutex_guard = mutex.lock().unwrap();
    let my_struct = mutex_guard.deref_mut();
    let x_ref = &mut my_struct.x;
    let y_ref = &mut my_struct.y;

    *x_ref += 1;
    *y_ref += 1;
}
```

Figure 5.9: A working version of the code in Figure 5.7.

The structure of the media framework element we translated is dictated to a certain degree by the open-source media framework used at Axis. This means that the suitability of an object-oriented structure for the element is closely related to how well the framework itself has been structured in this respect. To the extent that we used it, the framework seems to be a perfect fit for an object-oriented structure, much because of the way it represents different parts of an element. One of the ways it takes advantage of this in the Rust version is that traits are used to separate code into different sections based on which one of these element parts it is related to.

We found that the object-oriented structure made it easier to navigate and get an overview of the code, making it *easier to read*. We also found that this structure made the code focused and more in line with the single responsibility principle [34], which made it *easier to understand*. One example of this was the naming of framework functions. In C, the framework functions were usually named with the framework name first, followed by the element's name, followed in turn by the name of the actual function. In Rust, it is not necessary to include the framework name in the function name, as that information is instead conveyed by the module name. Similarly, the element's name can be left out, since the element is represented as a class that in turn contains the function. For instance, the C function `framework_element_function()` would correspond to `framework::Element::function()` in Rust.

Two Rust keywords that we found helpful in terms of readability are `let` and `fn`, which are used to declare variables and functions, respectively. The `let` keyword is placed in front of a variable declaration, while the `fn` keyword is placed in front of the function declaration. We found that this made it easy to distinguish between variables and functions. This can sometimes be slightly confusing in languages where variable declarations and function declarations start in the same way. It also made it possible to quickly find where a variable is declared or where a function starts, simply by searching for the keyword and the name.

Writeability

We found that the Rust compiler, `rustc`, was very helpful in terms of getting the code to work as intended. One of the reasons for this is that it *finds many bugs during compile time*, that may otherwise have been cumbersome to track down and fix by debugging. Another reason is that `rustc` presents *detailed error messages* in a way that makes it easy to understand the underlying problem. In addition to this, the error messages often include relevant suggestions for how to fix the problem, and where to look for more information about that particular problem.

Figure 5.10 and Figure 5.11 show the error messages generated when compiling similar programs written in Rust and C using `rustc` and `GCC`, respectively. These error messages reflect that there is a function that tries to return an integer value despite being declared not to do so. We find that `rustc` provides a more detailed and user-friendly description of the problem than `GCC`. First, `rustc` displays the name of the error first, indicating the type of error that has occurred. Second, the formatting of `rustc`'s error message makes the context clear, only showing the lines of code that are relevant to the error. Third, the expected type and the actual type are displayed, making the problem clear to the user. Finally, a reasonable fix is suggested and the user is given a command that can be run to get more information about that type of error.

```
error[E0308]: mismatched types
   --> src/main.rs:11:12
    |
 5  | fn no_return() {
    |                - possibly return type missing here?
 ...
 11 |     return a
    |            ^ expected (), found integer
    |
    = note: expected type ‘()‘
               found type ‘{integer}‘

error: aborting due to previous error

For more information about this error, try ‘rustc --explain E0308‘.
error: Could not compile ‘playground‘.

To learn more, run the command again with --verbose.
```

Figure 5.10: The output from `rustc` when compiling a program containing a function that tries to return an integer value despite being declared to not return anything.

```
src/main.c: In function ‘no_return’:
src/main.c:9:12: warning: ‘return’ with a value, in function returning void
     return a;
            ^
src/main.c:3:6: note: declared here
void no_return() {
     ^~~~~~~~~
```

Figure 5.11: The output from `GCC` when compiling a program containing a function that tries to return an integer value despite being declared to not return anything.

We also found that `rustc` sometimes displays only a few of all the error messages found. This helped tremendously in prioritising which errors to fix first.

Because of Rust's excellent error messages, it was often easier to *let the error message explain what to do* instead of looking it up in the documentation. This made it quicker to write correct code in certain situations. For instance, variable types are sometimes not visible in Rust code, such as when declaring a variable by using only `let` and omitting the type. This is very convenient most of the time, but in some cases the variable type must be known. An example of this is a function's return type, which must be specified explicitly. It is not always trivial to figure out which type to use in such situations, and this is where the compiler's error messages can be utilised. By simply omitting the return type, the error message will explain what type the compiler expected.

When we used Rust, we found that there are often *multiple ways to write code*. In particular, the ability to write both imperative-style code and functional-style code enabled us to choose the method that was most appropriate for the situation. For example, iterating over a list can either be done by using a standard for-loop with indices, or by using iterator methods (Section 1.5.1). On several occasions, we were very *satisfied with the Rust code* because we had been able to solve a problem in a way that we thought was clean and concise.

We found that the *Rust syntax helped us write correct code*. One example of this is the keyword `mut`, which must be used when declaring a mutable variable. To be able to mutate a variable, a deliberate choice has to be made by the programmer. We found that this encouraged planning ahead, and that it reduced the risk of us making accidental errors due to incorrect handling of variables.

We also found that Cargo increased writeability, mostly because of *Clippy* (Section 1.5.2). We found Clippy to be very helpful since it helped us keep the code as simple as possible.

We found that Rust's ownership system makes it easier to get correct code that is *guaranteed to not have certain issues*, e.g. memory leaks. However, as it does this by enforcing restrictions on how code is written, it can often *take longer to get the program to compile*. This time penalty of course depends on how used the programmer is to thinking about code in the way that the ownership system works. We found that the time penalty decreased as we became more experienced with Rust.

When we translated the plugin, we found Rust's guaranteed thread safety to be very helpful in terms of writing *correct parallel code*, but it did take a while to get there. As with the ownership system, there is also a time penalty associated with the compiler enforcing thread safety. It takes longer to get a program running, but it will most likely work as intended when it does compile. Something that the programmer needs to be aware of, however, is that the Rust compiler does not detect deadlocks.

Two specific Rust features that impacted writeability in a positive way were `Trait` and `Option<T>`. `Trait` helped us create an organised code structure, making it possible to put all related data and functionality together. This made it easy for us to decide where code should be placed based on which part of the plugin it affected. `Option<T>` helped us remember to check for missing values, as the Rust compiler will not accept code that does not handle the possibility of getting a `None` value.

Because Rust handles many things automatically, such as freeing memory, there is a lot *less boilerplate code* that needs to be written. This means that the programmer can focus on solving the problem instead of spending time with correctly handling the details. Because of this, we found that translating the plugin to Rust had the effect of making the code less complex in most cases. In addition to reducing complexity, having less boilerplate code also means that less code needs to be written in general.

Learnability

We started learning Rust with a background of having worked with several programming languages before, both low-level languages such as C and high-level languages such as Java. This meant that most of the basics in Rust were familiar, but certain language features such as the ownership system and the lifetime system were completely new to us. We were not used to thinking about code in terms of ownership, and neither did we have much experience handling data lifetimes.

During the translation of the media framework element, we never had to manually handle any lifetimes, but we were frequently "fighting the borrow checker"[1]. Getting our mental image of the ownership system to match with how it actually works was by far the greatest challenge for us in learning Rust. Had it not been for the Rust compiler constantly giving informative error messages, suggesting fixes for these violations of the ownership rules, it would have been even more difficult and time-consuming to learn the language.

Since the Rust compiler produces very informative error messages, we were constantly learning as we made mistakes. We found that we could get by relatively well by using the compiler as an "experienced programmer", who could suggest what to do when the program would not compile. However, in those cases where additional information or a more in-depth explanation was needed, the various resources that are available online were of much help. The three most important resources to us were the Rust book [50], the Rust standard library documentation [39], as well as the open-source media framework Rust documentation and examples. We found the Rust book to be an especially useful resource for learning the basics of the language.

The object-oriented structure in the Rust version of the framework resulted in some differences to the C version. As mentioned previously, one example of this is that function names were updated to reflect the framework being divided into modules, classes, and traits. This was mostly something positive for us in terms of learnability, as it had the effect of making it much easier to understand the structure of the framework.

We found getting started with programming in Rust to be very easy because of Cargo. Cargo makes it possible to create a project with the necessary settings and directory hierarchy with one command. Getting one's own code to run is just a matter of modifying the generated Rust source file, saving it, and then running it using `cargo run`. While there are flags and other options that can be set for a Rust project, knowing about them is not required when getting started. We appreciated how easy it was to set up a project using Cargo when we started to learn Rust, as it enabled us to focus on the language itself.

Knowledge base

During the code translation process, we found that the available knowledge base for Rust was good in some aspects, and bad in others. The book [50] [37] and the documentation for the standard library [39] were clear, up-to-date, and contained the necessary information.

[1]"Fighting the borrow checker" is an expression commonly used by Rust programmers to express their struggle to write code that follows the ownership rules.

However, when we tried searching online for help we found that solutions posted on forums (e.g. Stack Overflow) and blogs were often based on an older version of Rust. They were sometimes older than the official release of Rust, and did not work with the current version.

5.2.2 Developer perspective on programming languages

The interviews focused on the interviewee's experience with different languages. The focus was mostly on C, as all of the interviewees had experience with C and most of them use it on a daily basis. This covers topics such as what makes code easier to read, write, and learn. These results are presented based on the four factors in Section 4.1.2, i.e. Readability, Writeability, Learnability, and Knowledge base.

Readability

When we asked the employees about their thoughts on programming language readability, the questions were mostly targeting C, as that is the language they use the most. However, not all answers were directly connected to a specific language. Examples of this are factors such as *how the code is written*, following a *code style*, and *understanding what happens during code execution*.

Regarding C in particular, some said that they thought it could be hard to read and understand, giving examples of specific features or properties of the language that they felt impact the readability negatively. The two most frequently mentioned features were *macros* and *pointers*, but the C *syntax* was also something that some felt made the code difficult to read.

Macros were otherwise described as being useful, but in terms of readability they were often the first to be mentioned as something negative. One developer said that nested macros (macros calling other macros) with multiple levels could be particularly difficult to understand.

According to some of the employees, *pointers* can make the code difficult to understand. One employee said: "It can be confusing with all the stars[2], and if it's a pointer or not". Another employee said that when function pointers are involved, the code can be very hard to read. They explained that function pointers can make it difficult to understand the control flow of the program when debugging.

Many employees expressed that they thought readability has more to do with *how the code is written*, rather than it being something inherent to the language. One developer said: "It's more about praxis for developers, and you can always obfuscate anything with macros if you want to." One employee mentioned that developers sometimes write code that is hard to read because it happens to be more effective code than if it was written some other way.

Some employees suggested that it is important to have a *code style* to follow. One example was given about differences in indentation between different code standards used in various open source projects. They felt that this makes it difficult to read the code, highlighting the importance of following a single code style in a project. Another developer said: "It's all about setting up rules, and making sure that everyone follows the same rules." One employee

[2]In the C syntax, a star ($\star$) is used both to denote a pointer and to dereference a pointer. In some cases, multiple stars are involved, e.g. when having pointers that point to other pointers.

expressed that they did not think of C code as being hard to read, as long as conventions are followed.

Another type of readability is how easy it is to *understand what the code does*. In the interviews, this was one of the most frequently mentioned positive things about C. One developer explained that the lack of "hidden things" is something that can make C code easier to understand. Others expressed similar opinions, saying that C is a simple language and that nothing happens "behind the scenes".

Writeability

We learned from the interviews that one of the most appreciated features of C was that it enables the programmer to have much *control* over what happens. One employee expressed appreciation of the language's *simplicity* and the possibility to do exactly what you want in C. They explained that "everything you want to do that's a bit more complicated, you'll have to do yourself, and those are hopefully things you are knowledgeable about.". Another employee expressed similar opinions, saying that one of the reasons they love C is because you get to write everything yourself just the way you want it, and do not have to rely on methods created by others. Comparing C to Python, one developer said: "Python is definitely faster to get something going with, but you don't have the same control."

Another positive aspect that was mentioned about C was that it is a *powerful language*, making it possible to do things that are difficult or not even possible to do in other languages. An example of this, mentioned by one of the employees, was the ability to pass around function pointers, which would require the creation of objects in Java. One of the developers also pointed out that creating or controlling programs such as drivers or kernel software is easy to do in C.

While having *control* was generally regarded as a positive aspect of C, something that was mentioned by several employees was that it is easy to do "bad" things because of this control. One developer likened C to a scalpel, explaining that in the hands of a surgeon, a scalpel can do much good since it enables them to be very precise. However, there is also a possibility of doing much harm with it. The developer concluded that "you have to know what you're doing".

Some employees said that C can be *difficult to work with*. One developer said: "I have probably never encountered code that can be as difficult as C." Another employee argued that the *help* provided by C is limited when compared to Java. As an example, they mentioned the lack of standardised libraries in C, which was brought up by others as well. *Manual memory management* and the existence of *null pointers* were specifically mentioned as problematic aspects of C.

Making *mistakes* in C is very easy according to many of the employees, especially mistakes related to memory management. Because of this, many of them wanted more help from the language than C provides. One developer mentioned that language checks when casting from one type to another is something they miss in C. Others mentioned that having the language detect deadlocks would be very useful.

The developers also gave examples of features in other languages that they would appreciate, such as Java's garbage collector for automatic memory management. Employees mentioned the benefit of not having to search for memory leaks or worry about memory allocations. Another desired feature was to have *object-orientation* built into the language. One employee elaborated: "It's nice to have something that forces you to work in a more structured way, such as having an object-oriented approach. You have to create this yourself in C."

Learnability

The employees appreciated language properties in general that make the language easy to use and easy to understand. One example of this was that the language should be easy to learn or get started with. One of the developers mentioned that Python is easier to get started with than C, and other employees mentioned that Java was easier both to understand and to work with due to the existence of standardised libraries in Java.

Something that almost all of the developers had in common, was that they had used C for three or more years, with some even approaching twelve years. All but one of the developers said that C was their favourite programming language, and mentioned *habit* as one of the main reasons for this. One developer explained: "I guess that's what happens when it's the language you work with the most, so it's what I feel most comfortable with." One developer also said that they usually do not test new languages very often.

We asked the employees what they knew about Rust, and learned that most of them had heard about the language and knew at least a little bit about it and why it is popular. One of the reasons for this is that the media streaming framework used at Axis is transitioning to Rust themselves, and are actively encouraging their users to start transitioning to Rust as well. A few of the employees were quite interested in the language and said that they had looked at it more closely, with some of them even having written some code in the language themselves. One developer mentioned that Rust seems to be relatively *easy to understand*.

A few of the employees mentioned that the lack of knowledge about the Rust language could be a challenge when doing a transition from C to Rust. They argued that the developers would need to learn not only how to use Rust, but how to take full advantage of its features. One experienced C developer used the example that while developers who are used to working in C may be able to write programs in Python, that code will often look more like C code than idiomatic Python code.

Knowledge base

When asked about where they look for information or help, all developers said that they sometimes discuss the problem with *colleagues*, look for a solution on the *internet*, and read *documentation* (e.g. man pages or Application Programming Interfaces (APIs)). All of the developers noted that they use all of the previously mentioned sources, but not at the same time; the type of problem often decides what they will use. They only discuss it with colleagues if it is a larger problem, and for small problems (e.g. what function to use, or questions regarding the language) they often turn to documentation or internet forums for answers. Another developer mentioned that transitioning to Rust could mean that one has to get involved in the Rust community, which they had heard good things about.

A few developers mentioned that if they have questions about the code base, they sometimes use *tools* in order to browse through it. Another developer mentioned that they also like to use *books* as a source of information.

5.2.3 Focus group insights

The focus group participants did not express any negative opinions about any of the existing individual Programmer level factors. They did, however, comment that the three factors Readability, Writeability, and Learnability all affect each other.

One developer mentioned that *Readability depends on the way the code is written*, in addition to depending on the language itself. It was therefore concluded that it should be *easy to write expressive code* in the language. Error handling was mentioned as an example of this. In C, there are no built in structures for error handling, which means that such a structure must be created manually. This can lead to error handling logic being separated from the rest of the code, potentially making it harder to understand.

The developers expressed a desire for the possibility of *storing related information close together*, as long as it does not affect flexibility. More specifically, *exceptions* (a common structure for handling errors) was mentioned as a requested language feature. Other requested language features include *interfaces*, *garbage collection*, a *good standard library*, and *good free libraries*.

When it comes to learning a language, having *access to several examples* was something that the developers valued and appreciated. We also learned that *Learnability is an especially important factor*, since other departments read the code created at the Streaming department.

The developers highlighted the importance of having an *easy way to debug code* and an *easy way to test code*. This led to the suggestion of two new Programmer level factors: *Debuggability* and *Testability*.

5.3 Toolchain level

Here we present the results related to the toolchain level. From the programming we found how it is to use some Rust-specific tools such as Cargo [33], the Rust package manager (Section 1.5.2), and from the interviews we found what tools developers use in their everyday work.

5.3.1 Evaluation of Rust tools

During the porting process we were able to get hands-on experience using some tools together with Rust, which we cover here. These experiences include both tools specific to Rust and more general tools that can be used with other programming languages as well.

Debugging

During the porting process we did not need to use any tools for debugging, and the reason for this was twofold. First, the Rust compiler, `rustc`, is very strict and helpful, which enables finding and solving a lot of errors at compile-time. Second, we ported code, meaning that there is probably a lower chance for our code to contain bugs. At least there would probably be fewer bugs, since we had some working code to uses as a basis.

Compilation

One tool that is closely integrated with Rust is Cargo [33], Rust's package manager (Section 1.5.2). We used Cargo in order to handle our project's dependencies, compile the project with the appropriate flags, and to enforce a set of formatting rules. For these tasks we found Cargo to be very useful. We could easily add dependencies and specify how the project should be compiled by editing the `Cargo.toml` file, and enforcing formatting was easy with rustfmt [26].

At the time of writing, there is only one compiler for Rust, `rustc`. During our time using the compiler, mostly through Cargo, we found that the compiler helps the developer in writing correct code. The compiler helps the developer both by being able to detect a lot of errors at compile-time (also thanks to the programming language itself) but also by providing clear and helpful compiler messages.

We found integrating Cargo and Rust with the build system at Axis to be non-trivial. However, this is mostly due to us lacking knowledge about the build system. For someone with more experience, this might not be an issue. Thus, it is hard to estimate how much work it would take to integrate Cargo with the build system. This is also the reason why all our tests were run on the host computer, and not on the target camera.

Development

There are multiple editors and development environments that can be used together with Rust [23], e.g. Vim [44], Emacs [12], and Eclipse [8]. During the porting process we used only one of these, Vim, and we found it to work as expected with Rust. Vim does not support syntax highlighting for Rust "out of the box", but enabling it took little effort.

5.3.2 Common tools among developers

The interviews focused on what tools the interviewees used for different parts of the development process. This covers three of the factors from Section 4.1.3: Debugging, Compilation, and Development.

Debugging

When asked about what tools they liked to use for debugging, all of the developers said that they like to use the GNU Project Debugger (GDB)[3] for general debugging. Some developers also mentioned that they use debug prints for debugging. Some developers said that they like using Valgrind[4] for finding memory-related code defects.

There are also some tools that were only mentioned by one or two developers. However, it is likely that these (or similar) tools are used by the other developers as well.

[3]"GDB, the GNU Project debugger, allows you to see what is going on 'inside' another program while it executes – or what another program was doing at the moment it crashed." [11]

[4]"Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can *automatically detect many memory management and threading bugs*, and profile your programs in detail." [43]

The tools used by the Streaming department at Axis include: Helgrind[5], Atop[6], strace[7], gperf[8], AddressSanitizer[9], and MemorySanitizer[10].

Compilation

Most of the developers said that they did not have, nor were they aware if there was, any requirements on the compiler. Multiple developers noted that the compiler "should just work". One developer said that they want a compiler to be able to generate both unoptimised code that can be used for debugging, as well as code that is compact and optimised. The same developer also noted that they do not want to be forced to use a graphical environment in order to compile the code.

Development

For development, most developers preferred to use a more advanced text editor (e.g. vi [40], Vim [44], Emacs [12], or Atom [2]) since this allows them to add only the add-ons and extension that they want to use. One developer preferred to use Eclipse and noted that this was mostly due to habit, having used Eclipse for multiple years.

5.3.3 Focus group insights

During the focus group, we learned from the developers that *Debugging may be the most important factor on the Toolchain level*. Specifically, the developers *want debugging tools to support C*, and they also expressed a desire for already existing camera-specific debugging tools to *continue to work in a new language*. Being related to debugging to a certain extent, a *good test framework* was also desired by the developers.

In response to the Co-operation factor, it was mentioned that *co-operation is usually not language-dependent*. The developers concluded that it is not good if the language requires specific co-operation tools.

Another comment we received was that *developers rarely interact with the compiler directly*. The general opinion was that *compilers should just work*. However, build systems are often used, and this led to the suggestion of renaming the Compilation factor to Build system.

[5]"Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives." [16]

[6]"Atop is an ASCII full-screen performance monitor for Linux that is capable of reporting the activity of all processes (even if processes have finished during the interval), daily logging of system and process activity for long-term analysis, highlighting overloaded system resources by using colors, etc." [3]

[7]"strace is a diagnostic, debugging and instructional userspace utility for Linux. It is used to monitor and tamper with interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state." [31]

[8]"GNU gperf is a perfect hash function generator. For a given list of strings, it produces a hash function and hash table, in form of C or C++ code, for looking up a value depending on the input string. The hash function is perfect, which means that the hash table has no collisions, and the hash table lookup needs a single string comparison only." [15]

[9]"AddressSanitizer (aka ASan) is a memory error detector for C/C++. It finds: Use after free (dangling pointer dereference), Heap buffer overflow, Stack buffer overflow, Global buffer overflow, Use after return, Use after scope, Initialization order bugs, [and] Memory leaks." [1]

[10]"MemorySanitizer (MSan) is a detector of uninitialized memory reads in C/C++ programs." [20]

This way, the scope would be broadened and build system support could be something to consider when evaluating this factor.

There were some requested features related to development tools, including *code-completion*, *code skeletons*, and *something like Ctags*[11]. In general, the conclusion was that the developers were prepared to *compromise concerning development tools* if the language lacks support for their preferences. However, it was noted that it is almost a *requirement that Vim [44] can be used*.

5.4 Management level

The results from the programming focuses on our own experience with integration (M1), productivity (M3), and code defects (M4). From the interviews we instead found how the employees value integration (M1), OSS collaboration (M2), and code defects (M4).

5.4.1 Evaluation of Rust's impact on software projects

In this section we cover our experiences from the porting process regarding the Management level. This focuses on our experiences of Rust integration at Axis, how Rust affected our productivity, and how frequent different code defects were in our Rust code.

Integration

During the porting process, we received first-hand experience of integrating Rust with the environment at Axis. This included setting up Rust on an Axis workstation, using Rust with the tools used at Axis, writing a media framework plugin in Rust, testing Rust code using tests written in C, as well as running Rust code together with existing C code.

Setting up Rust and Cargo on the Axis workstation by following the steps provided in the Rust book [50] turned out to be very easy. We had no issues with installing Rust under the 64-bit Linux distribution used in the Axis workstations, and there were no problems with Axis-specific settings. As there are no requirements for using a specific IDE at Axis, there were no issues related to this either.

Writing media framework code in Rust was straightforward as well. First, the C version of the framework needed to be installed on the computer, and the Rust bindings had to be downloaded as well. Getting Rust to find the bindings was just a matter of including the file path to their location in the `Cargo.toml` file. It was then possible to import the framework modules into the code and start using their functionality.

While translating the element, we wanted to test the functionality that we were working on. We first thought that we had to translate the existing tests from C to Rust, but we soon realised that the tests could be used without translating them. By creating a simple shell script that built our Rust code and linked it to the C tests, we could run the tests as if we were testing the C code. The reason for this compatibility is that the tests did not communicate with the element directly, but instead did all communication via the framework. Having

[11]Ctags [6] is a tool that generates an index file that maps language objects (found in source code files or header files) to the location of their definition. The purpose of Ctags is to provide this information to programs such as text editors so that they can, for instance, locate the definition of a function.

access to working tests from the beginning was very valuable, and sped up the translation process significantly.

Setting up and running a pipeline with both C and Rust elements was easy to do. We made the element visible to the framework by first compiling the Rust code with Cargo, and then adding it to a framework path variable. After this, we set up a pipeline with already existing C elements, as well as the translated Rust element, and executed it without any issues.

Something we tested early on was to call C code from Rust code manually. It was difficult to find accurate and up to date information on how to do this, but we eventually managed to get it to work. We did this by using *cc*, which is a Rust library that builds C and C++ files and puts them in a static archive.

Productivity

Through learning and working with Rust, we found that the language has a number of features and properties that affect productivity in various ways. From our experiences, we feel that the positive aspects of Rust in terms of productivity outweigh the negative, at least in the long run. Overall, we feel that we had a higher productivity when working in Rust than we have previously had working in C.

One big reason for us having good productivity in Rust was the compiler, which through its *helpful error messages* enabled us to find and solve a lot of errors at compile-time. Because of this, *we never needed to debug our code*, which can be a very time-consuming task. Another aspect related to this is the strictness of Rust. Because errors are found during compile-time, they must be fixed before the program can be compiled and run. This means that *correct code can be obtained quicker*, but it also means that it *takes more time to get a program to run* than in other languages where not all errors are caught when compiling.

When encountering an issue in the code, the compiler's error messages were often sufficient to solve the problem, but sometimes we also needed to look up information elsewhere. In these situations, we could usually *find the information quickly* in the documentation. However, since Rust is a relatively new language, there were some *questions that we could not easily find answers to.*

Rust has many language features that we found to increase our productivity. Two examples of this are the features inspired by *functional programming* and *object-oriented programming*. These features made it possible for us to structure the code in a way that better corresponded to our understanding of how the program should work. It also *reduced the amount of details* that otherwise would have needed to be handled manually.

Other examples of Rust features that had a positive impact on our productivity are types such as `Result<T,E>` and `Option<T>`. These types provide a *structured way to handle errors and missing values*, which means that less time needed to be spent on correctly implementing such functionality manually.

Having a built-in package manager in *Cargo* was something we found to affect productivity in a positive way. When we needed to download a crate (package), we only needed to specify it as a dependency in the `Cargo.toml` file in order to make Cargo download it and make it available for use in our code.

The major issue for us in terms of productivity was Rust's *ownership and lifetime rules*. Even after having spent much time with Rust and reading through the excellent learning material, we often found ourselves in situations where we had difficulties in getting the code to work due to these rules. The compiler often suggested solutions to the issues we encountered, but it was still the most time-consuming aspect for us when writing Rust code.

Code defects

We encountered several code defects during the translation of the media framework element from C to Rust. Most of these were code defects related to Memory and Concurrency, and were all caught by the compiler, with one exception. Deadlocks were the only code defects that the Rust compiler did not warn us about.

Because we wrote concurrent code, deadlocks were not uncommon, and were definitely the hardest code defects to solve for us. It was easy to realise that a deadlock had happened, but figuring out where it had happened was not as easy. In our case, we had access to the working C code, which meant that we could relatively easily find where the deadlock had occurred.

We found that there were some tests made for the C version of the element that were unnecessary for the Rust version. One example of this is a test that checked if memory was leaked by counting the number of references to a piece of data. For the Rust version, these tests were not necessary as memory leaks are impossible[12].

5.4.2 The current impact on software projects

On the management level, the interviews focused on the employees' thoughts on the topics explained in Section 4.1.4. For integration, we asked the developers about their thoughts on both the possibility and the effects of a transition from C to Rust. The topic of OSS collaboration was investigated by asking about the current practices at the department. By asking about their thoughts on C, Rust, and current practices, we got an understanding of what the employees value in terms of productivity. We also asked the employees to share their experiences with code defects.

Integration

When asked about their opinion on the possibility of transitioning from C to Rust, most of the employees seemed positive, especially given the assumption that similar *performance* can be achieved using Rust instead of C. One developer mentioned that the only challenge they could see with a transition is performance. They elaborated by saying that if Rust can achieve similar or better performance than C, then there should be no noteworthy challenges with the transition. However, they also explained that if Rust cannot achieve the required levels of performance, a transition would be out of the question since the quality of the end product would deteriorate.

Some interviewees brought up the question of how to transition to a new language when working with a large code base. Several of the interviewees expressed that the focus should be on developing new code in Rust and ensuring *interoperability* with legacy code in C. Some interviewees stated that the need to rewrite code was the biggest challenge with transitioning

[12]Memory leaks are impossible to do in safe Rust, unless you explicitly call a function to do so: `std::mem::forget()`.

to a new language. One developer argued that since Rust is so compatible with C, they would not be restricted by the large amount of C code they are using today.

Regarding how a transition to Rust would affect other departments, there seemed to be mixed opinions. A few employees thought that other departments could be largely unaffected, as long as C and Rust can work together and the existing APIs can be used. A few other employees thought that other departments could be affected, for example in that they would need *knowledge* about Rust. Others were unsure about the extent to which other departments could be affected.

OSS collaboration

When asked about how their work is affected by Open Source Software (OSS), all of the employees said that it affects a big part of their work. This is mostly due to the fact that the department uses an open source media framework in most of their work, but also because some of the tools the developers use are open source.

The department also actively contributes to OSS, mostly to the media framework that they use, by uploading patches and reporting bugs. However, they are not maintainers so they do not review the contributions of others.

Some of the interviewed employees mentioned that a disadvantage of using OSS is that it sometimes takes a lot of time before the patch they submit is added to the software. This means that the department is sometimes forced to have local patches. A few employees mentioned that a benefit of using OSS is that the department does not have to write all the code (for the framework) and that bugs can be found and fixed much faster.

Productivity

From the interviews it was clear that the main reason for the employees having an interest in Rust was the positive things they had heard about the language. Most of the features they mentioned were in some way related to productivity, such as the language providing help with memory management and parallelism. One developer said that problems related to memory and parallelism take a long time to find and solve. They therefore argued that getting help from the language with these issues as early as during compilation could save a lot of time. As long as performance is not significantly affected, they thought that doing a transition to Rust based on an increase in productivity was not out of the question.

Code defects

When asked about code defects in C code, most of the employees said that concurrency and memory-related bugs are the most common. Some of the employees also mentioned performance-related code defects and crashes as common in C code. Most of the employees commented that concurrency and performance-related code defects are hard to solve, while only a few employees said that memory-related code defects and crashes are hard to solve.

5.4.3 Focus group insights

During the focus group, we learned from the developers that on the Management level it is important to *compare the long-term consequences with the short-term consequences*. All the developers agreed that this is especially important for the factor Productivity, which would most likely decrease directly after a transition but should increase long-term. The focus group also commented that *productivity could increase by using a higher-level language*, as this could result in a higher level of code quality.

It was also mentioned that Integration is very important, and that it must be possible to *use C and Rust together*. The reason for this is that they will most likely not rewrite the existing C code to Rust if a transition would be made, and instead focus on writing new code in Rust and integrating it with the existing C code.

The focus group also considered Code defects to be an important factor and commented that it is important that new code defects are not *introduced as a result of the transition*. The developers also mentioned that being able to make *guarantees* about the types of possible code defects could be very useful.

Regarding OSS Collaboration, the focus group concluded that this could be affected in both positive and negative ways. Since the open-source media streaming framework used by the department is transitioning to Rust there is a possibility that, if the department want to contribute to the framework, then these *contributions would be given higher priority* by the framework maintainers. On the other hand, as there are fewer framework maintainers working with Rust, it may *take more time for them to review contributions*.

The focus group also mentioned that possibilities for *education* is important, and that using Rust could affect *recruitment* and how attractive Axis is on the job market. Another important factor mentioned by the focus group was how time needs to be allocated between writing new code and *maintaining* the existing code. Thus, three new factors were suggested for the Management level: *Education*, *Recruitment*, and *Maintainability*.

Chapter 6

Discussion

In this chapter we discuss the results from the previous chapters. This focuses mostly on the results covered in Chapter 5, the results related to the Programming Language Transition Framework (PLTF) (Chapter 4), and our methodology (Chapter 3). This discussion is divided into three sections, based on our research questions (Section 1.3.1). Each section is then divided into four subsections, one for each PLTF level.

6.1 How to evaluate a transition from C to Rust (RQ1)

We found that almost all of the factors in the PLTF are important during a programming language transition. However, there are some factors that may need a more detailed definition and there are also additional factors that should be investigated. A summary of the PLTF, including suggestions from the focus group, is presented in Appendix C.

One of the levels, the Toolchain level, did not get any suggestions for additional factors, indicating that it is more complete. The Execution level, Programmer level, and Management level got one or more suggestions for additional factors, indicating that there may be even more factors missing from them. All levels were seen as relevant, and the factors on each level were considered to be mostly affected by other factors on the same level, indicating that the PLTF has a good level classification and that none of the factors should be moved to another level.

While we developed the PLTF with Axis in mind, we argue that it is useful in other cases as well. We base this on the fact that our motivations for including the different factors (Section 4.1) are not specific for Axis or the given use-case. However, some factors can be more or less important based on the context. Performance is very important in embedded systems, but in other domains this may not be the case. Other factors, such as Productivity and Readability, are more general and less dependant on the context.

6.1.1 Execution level

When evaluating the two languages on the Execution level, we found that porting code from one language to the other, and then comparing the two programs, worked well. This process also *saved us a lot of time*, compared to having to start writing Rust code from scratch.

During the focus group we found that developers are not only interested in *how much memory a program uses*, but also *how memory is used* (Section 5.1.3). This is not unreasonable as how long time it takes to read from and write to the memory can affect the overall execution time of the program.

As mentioned during the focus group, there are some external factors that can affect Execution level, namely *language culture* and *product specification* (Section 5.1.3). While these can affect the PLTF factors on the Execution level, they were regarded as too specific (in the case of product specification) or too hard to measure (in the case of language culture) to warrant being their own factors.

6.1.2 Programmer level

We found that porting code is not as useful as writing code from scratch when evaluating usability. The reason for this is that some usability aspects are only present when writing new code. However, we still found that we could evaluate a great deal of the language's usability through the process of porting C code to Rust.

When evaluating the Programmer level factors, we found that they are vague and difficult to investigate in a structured way. In particular, more detailed definitions and methods for evaluating the factors would have made the process easier. We attribute this in part to the factors being relatively *open in nature*, as well as there being some *overlap between the factors*. For instance, one aspect of Readability may also affect Writeability.

As was mentioned during the focus group, readability is not only dependent on the language, but also *dependent on the author* (Section 5.2.3). The language may be able to increase the code's readability, but a developer may still be able to decrease the readability by writing the code in a way that obfuscates the code's functionality. If the readability is affected by both the language and the developer, then it would be interesting to investigate what the ratio between these two are. How much does the language affect readability and how much does the developer affect readability (or how much do they need to affect it)?

What would make one programming language more readable than another? A programming language may be able to increase the readability on a small scale (e.g. what a line of code does or how the standard library works), by matching the mental image of the reader. On a larger scale (e.g. what a function or program does), then more of the responsibility in terms of readability falls on the developer in that they have to structure their code in a way that is logical to the reader. Readability (and in some part writeability) could perhaps be measured by how much work the developer has to do in order to make the code readable, and to what extent the language encourages writing readable code.

When evaluating the usability of a programming language, it is important to not only evaluate the more obvious factors (such as Readability and Writeability) but also how it is to work with the code in other ways. During the focus group we found that developers were also interested in *Debuggability* (how easy it is to debug the code) and *Testability* (how easy it is to test the code) (Section 5.2.3). This seems reasonable, since software development is not only about reading and writing code, but also checking that it works as intended and correcting potential bugs.

6.1.3 Toolchain level

The Toolchain level can be evaluated by investigating if the *currently used tools work with the new language*. Since having to learn to use a lot of new tools during a transition could result in a *too big barrier of entry* to the new language, it may be preferred to continue using the currently used tools even after a transition. However, if the new programming language would open up possibilities to new, perhaps better tools, then evaluating these tools could be important.

As we found that *tools for co-operation are often language independent*, this factor may be unnecessary (Section 5.3.3). As long as the new language does not put restrictions on what co-operation tools could be used, then there is probably no need to investigate further. If the new language would require special tools for co-operation, then that could be a deal-breaker as it would mean that you have to use one co-operation tool for the new language, and another for the old. However, such a scenario should probably be seen as an exception, and because of this, *the Co-operation factor should be mostly irrelevant*.

6.1.4 Management level

When evaluating the Management level, it is important to *evaluate both short-term and long-term*, as some factors may affect both in different ways. It is safe to assume that a lot of the factors on the Management level would not show an immediate gain when transitioning. It is more likely that they would yield a loss in the beginning, as having to spend time and other resources on educating the workforce is a direct cost – or an investment – that could potentially pay off later.

Productivity is one example of a factor that will likely be affected negatively in the beginning, which would be considered a short-term cost. The language's benefits in terms of productivity will therefore not be noticeable until some time later, potentially resulting in a long-term gain. Because of this, it may be beneficial to do a *small-scale experiment* with a group of individuals who will learn the language and start using it. The results from this experiment could then be used to evaluate the Management level impact on a larger scale.

In this particular case study, it was clear that the focus for the Streaming department was not to translate already existing code. This means that a *factor about code translation* was not necessary. However, in other cases where code translation will occur, this Management level factor would be of great importance. For example, having *automated tools* for this kind of task has the potential of dramatically decreasing the time it takes to translate code, and can therefore be important to consider when making a decision about a potential language transition.

During the focus group, we received suggestions for three more factors for the Management level: *Education*, *Maintainability*, and *Recruitment* (Section 5.4.3). Since this level only has four factors, adding three new factors would equal a 75% increase. This indicates that there may be *additional factors missing from the Management level*.

6.2 Consequences of transitioning from C to Rust (RQ2)

We found that transitioning to Rust can result in both positive and negative consequences. On the Execution level, we found that for most of the factors Rust can perform on par with C, performing better than C in some tests and worse than C in other tests. On the Programmer level, we found that Rust is able to provide a better developer experience than C, by offering clearer error messages and more ways to write the code. On the Toolchain level, we found that there is little difference between Rust and C, as most tools support both languages. Finally, on the Management level it is hard to draw a conclusion. For some factors (e.g. Code defects), we found that Rust is better than C, while for other factors (e.g. Productivity) it is hard to know how the department would be affected.

6.2.1 Execution level

As we see in Section 5.1.1, the *Rust versions of the elements have longer execution times* than their corresponding C versions. This is true both in the case of the element translated by us and in the case of the element written by the developers of the framework. The Rust version of the elements have an average execution time that is 4.24% (0.18 seconds) and 0.09% (0.0004 seconds) longer than the corresponding C version, respectively. However, these measurements are based on measuring the execution time of the entire pipeline, meaning that they include more than just the execution time of the element, which is likely to be a small part of the entire pipeline.

When we measured latency, we found that the Rust version of our translated plugin has, on average, a 25.69% (5.00 microseconds) longer latency than the C version. While we were unable to get any conclusive results concerning the reason for this difference in latency, we did get some indications using the performance analyser tool `perf` [21]. One function in the Rust code showed up in several measurements, indicating that it is among the most time-consuming functions in the program. Although this perhaps does not account for the entire difference in latency between the two versions of the element, it certainly could be one of the major causes. This is especially likely as the function did not show up in the measurements for the corresponding C version.

Because we used the original C tests to test our Rust version, we were limited by how they tested certain aspects of the element. The way one of these aspects was tested made it impossible for us to follow the code style used in the Rust plugins made by the media framework, and instead forced us to write sub-optimal code in this function. The main difference between the Rust implementation and the C implementation of this function is that the Rust version uses a hash map data structure, while the C function utilises functionality built into the media framework. Our assumption was that using a hash map would improve performance, but as the measurements indicate that this function is slower in Rust, it may be that this approach is actually slower. Unfortunately we lacked the time to test if implementing the C approach in Rust would make it faster, or if the performance difference is due to something else.

Another potential reason for the performance difference is that we were not able to fully utilise the media streaming framework, as we are not as well-versed on that subject. However, since we did not write any new functionality but instead ported existing C code to Rust, this factor should be fairly limited.

When we compared the average latency difference between the C and Rust version of a plugin written by the developers in the media framework, we found that the C version had a 30.30% (4.11 microseconds) longer latency than the Rust version. These results indicate that there is either something special about the particular plugin we translated that makes Rust less suitable for it, or that our code translation is sub-optimal. Because of the measurements indicating that at least one function may be implemented sub-optimally, we believe that the latter is most likely.

From the related works by Rikte, Light, and Wilkens, we see that the performance difference between C and Rust varies in terms of execution time (Section 2.2). This pattern is also reflected in our results, where in one case the Rust version has a 25.69% longer latency, and in the other case the C version has a 30.30% longer latency. From these results, we can conclude that the execution time of Rust can be either significantly faster than C, or significantly slower than C. This suggests that there may be some aspects of the Rust programming language that make it better in some situations, and some aspects that make it worse in others.

When comparing the memory usage we see that our Rust version of the element has, on average, a 1.656% (177 kbytes) greater maximum memory usage than the corresponding C version. However, comparing the Rust version and C version of an element written by the developers of the framework we see that the C version has, on average, a 43.71% (3129 kbytes) greater maximum memory usage then the corresponding Rust version. When Rikte ported a Linux daemon from C to Rust, he found that the Rust version used 7% (3MB) more memory than the C version (Section 2.2). These results show Rust using significantly less memory than C in the best case, while using only slightly more memory than C in the worst case.

Because of the differences in results, and due to the uncertainty concerning the cause of these differences, it is difficult to draw the conclusion that Rust is better or worse than C when it comes to latency and memory usage. It is important to keep in mind that, while the performance difference may correspond to the difference between the C language and the Rust language, it is also dependent on the particular implementations. It is clear that *Rust has the potential to perform better than C*, but we can also see that *translating code from C to Rust does not necessarily result in equal or better performance.*

6.2.2 Programmer level

One of the most positive aspects of Rust, in our opinion, is that so *many errors are found at compile time.* We found that we always felt confident when the program compiled. This confidence came from knowing that successful compilation means the absence of certain code defects, making it more likely that the program would work as expected. We believe that this guarantee concerning code defects is of great importance to a developer. This is because those code defects can be ruled out as potential reasons to their code not working correctly, which limits the number of possibilities to investigate.

Some compilers produce errors that can take some time to understand, especially for those who do not have much experience working with the compiler or the language. One of the reasons for this is that they present the error from the compiler's perspective, which can be different from the programmer's perspective. The first part of solving the problem becomes interpreting the error, and when the programmer finally understands the error, they can start investigating what causes the problem and start solving it. We found that `rustc`, the Rust compiler, provides *detailed error messages* that give helpful advice on how to resolve the error (Section 5.2.1). This is in line with the experiences reported by Rikte, namely that compiler messages were found to be user-friendly (Section 2.3).

We found that Rust's *error messages could be used as a way to obtain information* on how to write code (Section 5.2.1). We are used to working in languages that mostly report what is wrong without giving suggestions on how to fix the issues, so this was something completely new for us. Being able to utilise this method was very helpful, since it gave us the information we needed to solve our specific problems. Finding the same information in documentation or online forums is sometimes difficult, since the programmer is first required to understand the issue to some extent in order to know where to look or what to look for. Another issue is that the problem needs to be expressed in relevant keywords for a search to yield useful results. By letting the compiler provide the information instead, these issues are avoided. Granted, not all problems can be solved in this way, as it is limited to syntax-related issues and also because it relies on the compiler correctly interpreting what the programmer intends to do. Nevertheless, we found this to be a very useful feature, and since Cargo allows checking the source code for errors without actually compiling anything[1], this should in theory be feasible even for large projects.

In the field of user experience, it is beneficial to have many possible paths to accomplish the same goal, since it increases the chance that the user finds a path that is natural to them in their current situation. We argue that this holds true for programming languages as well, in that having multiple ways to write code that does more or less the same thing can make it more likely that the programmer finds an approach that is natural for solving the problem. This has the potential to positively affect both readability and writeability if it makes the code convey the progammer's intent more clearly. We found that this usually holds true for Rust, as *we could often choose a way to write the code that was natural to us* (Section 5.2.1). We therefore agree with Light's assessment that Rust allows the meaning of code to be expressed in a richer way than is possible to do in C (Section 2.3). Depending on the situation, we also agree with Rikte's opinion that a functional style has a positive effect on the readability of Rust (Section 2.3).

Being able to write Rust code in a way that was well-suited for the situation also had the effect of making us more *satisfied with the code* in general. We started to like Rust early on during our study, which we attribute much to the fact that we find its language features to be very clean and efficient ways of writing code. According to Meyerovich and Rabkin, enjoyment of a language is highly correlated to expressiveness, elegant code, and the ease of abstracting patterns in code (Section 2.3). Our experiences with Rust support this claim.

[1]Checking the source code for errors without compiling can be done in Cargo by running `cargo check`.

6.2.3 Toolchain level

On the toolchain level we see that Rust offers certain benefits over C, with the two major ones being Cargo (Section 1.5.2) and the Rust compiler.

While Rust only offers one compiler, there are multiple compilers available for C code. However, for this comparison we will limit ourselves to only looking at the C compiler in the GNU Compiler Collection (GCC) [10], which is arguably one of the more commonly used C compilers. Comparing these two we find the *error messages given by the Rust compiler to be much more helpful than GCC's error messages* (Section 5.2.1 and Section 5.3.1). We attribute this to the fact that the error messages from `rustc` is more verbose and also gives a suggestion for how to fix the error. This does not only help the developer with writing correct code, but can also decrease the time needed to solve the error. Thus, `rustc` is, in addition to being a functionining Rust compiler, a tool that *actively increases the writeability of Rust and the developers productivity*. This was also the case for Wilkens who found that Rust offers "excellent tool support", which contributed to the lower development time of Rust, compared to C (Section 2.4).

Based on the interviews, we know that a lot of the developers like using the GNU Project Debugger (GDB) [11] and Valgrind [43] for debugging. Although Rust does not offer better support for these tools than C, it should be noted that there is a specific version of GDB for Rust called *rust-gdb* and that it is possible to use Valgrind together with Rust. While we did not use any of these tools, as we did not need to debug our code, Rikte tested them and noted that the *they work as expected* (Section 2.4). Thus, if a transition to Rust would be made, then the most used tools for debugging would likely be unaffected.

Regarding Integrated Development Environments (IDEs), one developer preferred using Eclipse while most of the developers preferred not using an IDE but instead use a more advanced text editor (e.g. vi/Vim, Emacs, or Atom) (Section 5.3.2). As Karl Rikte mentioned, there is a Rust extension for Eclipse (and IntelliJ) and it works well, although some features may be missing (Section 2.4). Obviously, the text editors have support for writing Rust code, but more importantly they also have extensions available that can provide features such as syntax highlighting and code completion. While we did not test all the editors mentioned during the interviews, we did get first hand experience using Vim, and can confirm that it *works as expected and that enabling syntax highlighting was trivial* (Section 5.3.1). This was the most important editor to test, as the focus group concluded that it was nigh a requirement that Vim is supported (Section 5.3.3). Thus, if a transition to Rust would be made, the tools used for development would most likely be unaffected.

While a lot of tools are language independent or have extensions for Rust, there could still be some negative consequences of transitioning from C to Rust. As the developers are free to use the tools they like, there is a risk that some of the developers are using tools that neither are language independent nor have Rust versions. This will most likely have only a small impact, as most of the tools used are language independent or there exist similar tools for Rust.

6.2.4 Management level

Rust offers a few features that could be beneficial on the management level, as it can be integrated with C and potentially reduce the frequency of some code defects.

As we were able to use the unit tests (written in C) for the C version of the element to test the Rust version we can say that *integrating Rust with C is possible to a certain extent* (Section 5.4.1). Thus, a transition to Rust would not require one to rewrite all the original tests, which in turn would make a potential transition easier. This is not so much a benefit of Rust over C, but more so a benefit of Rust in general.

Another benefit of Rust is its focus on correctness. Since the language and its compiler have this focus on correctness, *the frequency of some code defect can be reduced and some code defects may be completely eliminated*. During the porting process, the only code defects that we encountered during runtime were concurrency-related (i.e. deadlock) which was due to the fact that the Rust version of the element was not fully implemented at that time (Section 5.4.1). However, the act of porting existing code from one language to another would most likely result in fewer code defects compared to writing entirely new code.

Also related to Rust's focus on correctness is the possibility of finding errors in existing code. Rewriting code in another language usually opens up the possibility of introducing errors. Rust's strictness will not only reduce the risk of this happening, but will also make the translation process a method for finding previously unnoticed errors in the code. While we did not find any errors in the C code that we ported, this was the case for Rikte (Section 2.3). Rikte does not mention whether the error in the C code was found through carefully reading the C code, or if it was found with the help of Rust and `rustc`.

Integrating Rust into the existing environment could be difficult in some regards. Based on our own experience from the porting process, we know that *incorporating Rust into the build system was nontrivial* (Section 5.3.1). We were unable to do this mostly due to the fact that we lacked knowledge about the build system, but it still shows that this is a hurdle that must be overcome.

Regarding productivity, it is hard to know for certain how the department would be affected by a transition. We felt that our *productivity was higher in Rust* than our previous experiences of using C (Section 5.4.1). We attribute this increase in productivity in large part to the helpful error messages from the compiler, but also to some of Rust's language features, such as `Result<T,E>` and `Option<T>`. However, since we ported code instead of writing it from scratch, we would most likely get a feeling of being more productive. The reason for this is that we always had working C code to use as a basis for our design and our solutions.

Wilkens also found that they were more productive when working with Rust than when working with C, and attribute this to the tool support for Rust (Section 2.4 and 2.5). However, it is hard to say if it is universally true that developers are more productive in Rust than C. What we can say is that if a transition would be made, there will most likely be a decrease in productivity during a period of time, as developers will need time in order to learn Rust and how to fully utilise its features.

6.3 Challenges of transitioning from C to Rust (RQ3)

We found that each level has their own challenges and that we cannot conclude that one challenge is more important than another, they are only different. One challenge may be easier to overcome than another, however if it cannot be overcome then it is just as limiting as any of the other challenges.

6.3.1 Execution level

One important challenge that we found on the Execution level is the *size of executable for Rust*. During the interviews we found that several employees regarded size of executable, along with execution time and memory usage, as one of the more important factors (Section 5.1.2). While our execution level results from the porting process (Section 5.1.1) show that Rust can perform on par with C when it comes to execution time and memory usage, the same could not be said for size of executable. While we were unable to properly investigate the difference in size of executable between our Rust version of the element and the corresponding C element, we were able to get some preliminary results that indicate that the Rust version has a significantly larger binary size than the C version. Similar results were also found by Karl Rikte, who did a more thorough comparison of the binary size of C and Rust (Section 2.2).

6.3.2 Programmer level

The biggest challenge we experienced on the Programmer level is that *Rust has a steep learning curve*, which means that it will take time for developers to learn to use Rust effectively. While there exist excellent learning resources, it still took some time for us to learn Rust, with some of Rust's features (e.g. Ownership) being harder to learn than others (Section 5.2.1). The steep learning curve of Rust, especially for the borrow system, was also mentioned by Rikte (Section 2.3).

6.3.3 Toolchain level

The only challenge we encountered on the Toolchain level was *integrating Rust with the build system*. These problems were due to our lack of knowledge about the build system at Axis, and that we did not have time to learn more about it (Section 5.3.1). We believe that it is possible to integrate Rust with the build system at Axis, however it should be done by someone with more knowledge about the build system.

6.3.4 Management level

The most important challenge on the Management level is that *it will take time before the transition would yield a positive result*. In the beginning of a transition it is safe to assume that there will be a drop in productivity, as developers will need time to get comfortable with the language. How long it will take for the transition to yield a positive result can be hard to estimate, and thus it is important that the department is able to handle a temporary decrease in productivity, if a transition would be made.

6.4 Threats to validity

Just like any other study, our study has its limitations. Here, we will discuss these limitations and how we tried to mitigate some of them, based on the guidelines for software engineering-related case studies proposed by Runeson and Höst [56].

6.4.1 Construct validity

Our choice of methodology in evaluating aspects of a Rust transition by *porting code rather than writing new code*, poses a threat to construct validity. When porting existing code that is known to work, it is tempting to use the exact same structure instead of creating a structure that best fits the new language. As for C and Rust, the two languages are rather similar, meaning that it is relatively easy to use a C-like structure in Rust code by making only minor modifications. This usually does not take full advantage of the Rust language. In our case, however, we had access to examples of other plugins written in such a way as to take advantage of Rust's strengths, making it easier for us to do so as well.

Since the Streaming team is not primarily interested in porting old code to Rust, but rather using the language for new projects, our evaluation is limited as *it does not directly correspond to their use case*. However, it is likely that new projects will draw inspiration from the structure of already existing Rust plugins within the media streaming framework. The structure we used will likely be used by Streaming as well, meaning that our evaluation of using that structure is relevant to their use case.

6.4.2 External validity

Seeing as most of our study focused on Axis, and its Streaming department, our results *may be hard to generalise* and apply to other cases. This is perhaps most noticeable in the results from the porting process, interviews, and the PLTF.

When we ported C code to Rust we did so by porting a media streaming element from C to Rust. This means that *our experience is very focused around using Rust together with the media streaming framework*. It also means that the performance results we obtained by comparing C and Rust are based on using them in an embedded context to create media streaming applications. Thus, in other cases a comparison between C and Rust may yield different results.

During the interviews, we *interviewed only eight employees*, six developers and two managers, all working at Axis. Thus, the interview results regarding what developers think about C might not accurately represent what C developers in general think about the language.

We *designed the PLTF with Axis' needs in mind*, particularly the Streaming department, and based on feedback from our supervisors at Axis. This could result in the PLTF having factors that are relevant for Axis but not other companies. Likewise, the PLTF could also be missing factors that are important in other cases. However, this is partly mitigated by the fact that we were able to relate our factors to factors mentioned in other works.

6.4.3 Reliability

Early on in the study, when we learned Rust, *we found that we liked a lot of the features that the language offers*. While this can be seen as a big plus in favour of Rust, it also means that we could have been biased towards Rust to some extent from an early stage, which could impact the reliability of our results.

Both authors were *inexperienced with conducting interviews and focus groups*, which means that there is a risk that some information was misinterpreted. We tried to mitigate this by applying triangulation, both between the authors and towards the interview participants. We also sent the interview transcripts to the interviewees for validation. Each output produced by one author, i.e. transcription, coding, and reporting, was reviewed by and accepted by the other author.

The participants of the *focus group were selected based on a recommendation by Axis* in terms of their roles and potential to contribute to the discussion. This resulted in a group where half of the participants had also been part of the interviews. As their opinions and insights had already been used to shape the PLTF, the amount of new information that could be gathered was limited.

Due to time constraints, we were *unable to transcribe the recordings from the focus group*. As we were aware of this during the focus group, we decided to take notes, which we periodically validated with the participants during the focus group.

During the interviews, we noticed that our *interview guide targeted developers*. The interview guide had some questions that managers were not able to answer, and it was also missing some questions more suitable for managers. This resulted in that we, during the interviews, tried to rephrase some questions so that they became more suitable for managers. Instead of asking a manager "What tools do you use?" we asked "What tools do you think the developers like to use?", thus still getting a valuable answer.

Due to time constraints we *were only able to interview eight employees*, which means that there is less certainty in the interview results. It was also not possible for us to cover the entire PLTF during the interviews, as some factors (e.g. how easy it is to learn, read, and write Rust code) are hard to evaluate during an interview. Thus, we instead asked the interviewees what they think would affect these factors and, since all the developers had experience with C, what they think of C with regards to these factors.

Chapter 7
Conclusions

In this chapter we summarise this study as well as cover some ideas for future research.

7.1 Summary

The Programming Language Transition Framework (PLTF) that we constructed *worked well as a basis for investigating a programming language transition*. The focus group concluded that all of the included factors are important and that the four levels are a good way of classifying the factors. However, the PLTF still needs work, as we during the focus group got suggestions for more factors that they think are important.

On the execution level, we had *mixed results*. Rust is able to perform better than C for most of the factors, but this is not guaranteed for all cases. The performance is not only dependent on the application but also on how the code is written (e.g. which code style or data structures are used). When it comes to the size of the executable for the two languages, C seems to produce an executable with a smaller size. While we were unable to thoroughly investigate this claim, it is supported by Karl Rikte's study [55].

On the programmer level, we found that *Rust provides a better user experience than C*. This is partly because of its high-level features inspired by object-oriented programming and functional programming, but also because it handles things such as memory management automatically. While C is a powerful language, it is also a language that requires a lot of discipline, especially when it comes to memory management and pointers. Rust is able to give the programmer the same level of control as C, but without forcing the programmer to have the same level of discipline. The reason for this is that the Rust compiler is able to detect a lot of errors at compile-time (errors that would be detected at run-time in C) and often gives helpful tips on how these errors can be fixed. Rust is able to find these errors by using a special ownership system, which dictates how and when data can be used. In practice, this means that the Rust compiler will reject a lot of code that would compile in other languages. Since these ownership rules are not enforced in most programming languages, they make the learning curve of Rust rather steep even for those used to writing code in said languages.

On the toolchain level, we found that *most tools that support C also support Rust*. Some of the more commonly used debugging tools for C (GDB and Valgrind) have either support for Rust out of the box or have a separate version for Rust. Some tools, e.g. tools for collaboration and more advanced text editors, are often language-independent and will thus work without problems in both languages. In order to get syntax highlighting in some editors, extensions may need to be installed, but this is often trivial. In addition to being able to use Rust with many existing tools, Rust also comes with its own tools. The most important of these is Cargo, the Rust package manager, which is a great tool for handling Rust projects. It allows one to easily set up, compile, and handle Rust projects while also providing possibilities to install helpful extensions such as Clippy. The biggest challenge we encountered on the toolchain level was that we were unable to integrate Rust with the build system at Axis. However, this is because we were not familiar with the build system, which time restrictions prevented us from investigating further.

On the management level, we found that Rust can be used together with C and that Rust can help reduce the number of code defects. Using *C and Rust together worked without any major problems*. This was mostly because the media framework supports the use of both C and Rust elements in the same pipeline. Rust's ownership system makes it so that a lot of errors are found at compile-time, which means that the risk of there being code defects in the finished code is lower. *Using safe Rust, it is possible to make guarantees about certain code defects*, e.g. that there are no memory leaks. Once we were used to the ownership system, we found that our productivity when using Rust was high, which was also the case for Wilkens [58].

On the one hand, we found the Rust developer experience to be superior to C, with high-level features and a reduced risk for introducing bugs. On the other hand, the performance measurements showed mixed results, with Rust performing better in some cases and worse in others. Thus, we cannot say that Rust is strictly better than C, or vice versa. However, we can conclude that the four-year-old newcomer that is Rust is a good alternative to the forty-seven-year-old veteran that is C.

7.2 Future research

One of the big parts of this study was the construction of our PLTF. The motivation behind the factors included in the PLTF are based on the results from our literature study and input from Axis, making the PLTF specialised for our case. The PLTF could most likely be used in other cases as well, but might need to have factors added or removed from it. Thus, *a study that evaluates the usefulness of our PLTF in other cases* could be interesting.

While we were unable to run Rust on target, which is due to our lack of knowledge about the build system at Axis, we believe that it is possible. *A study that compares the performance between C and Rust, when running on target* (or other embedded systems) could be interesting.

While we were able to present some findings on the Management level, a lot of the factors on this level need to be investigated over a longer period of time. It would therefore be interesting to see the result from *a study that focuses on the Management level*, and investigates these factors during a longer period of time.

One of the biggest drawbacks we found with Rust was the steep learning curve, which was also mentioned by other studies. In all of these studies, including our own, the authors already had experience with other programming languages. It would be interesting to see *how the learning curve for Rust compares to the learning curve for C, given no previous programming experience.*

Bibliography

[1] AddressSanitizer. `https://github.com/google/sanitizers/wiki/AddressSanitizer`. Accessed: 2019-05-28.

[2] Atom. `https://atom.io/`. Accessed: 2019-05-28.

[3] Atop. `https://www.atoptool.nl/`. Accessed: 2019-05-28.

[4] Axis - History. `https://www.axis.com/about-axis/history`. Accessed: 2019-06-18.

[5] Clippy. `https://github.com/rust-lang/rust-clippy`. Accessed: 2019-06-07.

[6] Ctags. `http://ctags.sourceforge.net/`. Accessed: 2019-10-17.

[7] DuckDuckGo. `https://duckduckgo.com/`. Accessed: 2019-04-29.

[8] Eclipse. `https://www.eclipse.org/`. Accessed: 2019-06-07.

[9] Evaluating Programming Languages. `https://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html`. Accessed: 2019-10-12.

[10] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. Accessed: 2019-05-14.

[11] GDB, the GNU Project Debugger. `https://www.gnu.org/software/gdb/`. Accessed: 2019-05-24.

[12] GNU Emacs. `https://www.gnu.org/software/emacs/`. Accessed: 2019-05-28.

[13] Google. `https://www.google.com/`. Accessed: 2019-04-29.

[14] Google Scholar. `https://scholar.google.com/`. Accessed: 2019-04-29.

[15] gperf. https://www.gnu.org/software/gperf/. Accessed: 2019-05-28.

[16] Helgrind: a thread error detector. http://valgrind.org/docs/manual/hg-manual.html. Accessed: 2019-05-28.

[17] IEEE Xplore. https://ieeexplore.ieee.org/. Accessed: 2019-04-30.

[18] ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. Accessed: 2019-10-12.

[19] LUBsearch. http://lubsearch.lub.lu.se/. Accessed: 2019-04-29.

[20] MemorySanitizer. https://github.com/google/sanitizers/wiki/MemorySanitizer. Accessed: 2019-05-28.

[21] perf. https://linux.die.net/man/1/perf. Accessed: 2019-10-17.

[22] Rust – Production users. https://www.rust-lang.org/production/users. Accessed: 2019-10-10.

[23] Rust – Tools. https://www.rust-lang.org/tools. Accessed: 2019-06-07.

[24] Rust blog: Announcing Rust 1.0. https://blog.rust-lang.org/2015/05/15/Rust-1.0.html. Accessed: 2019-06-21.

[25] Rust blog: Non-lexical lifetimes. https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html#non-lexical-lifetimes. Accessed: 2019-09-19.

[26] rustfmt. https://github.com/rust-lang/rustfmt. Accessed: 2019-06-07.

[27] StackOverflow Developer Survey Results 2016. https://insights.stackoverflow.com/survey/2016/. Accessed: 2019-05-27.

[28] StackOverflow Developer Survey Results 2017. https://insights.stackoverflow.com/survey/2017/. Accessed: 2019-05-27.

[29] StackOverflow Developer Survey Results 2018. https://insights.stackoverflow.com/survey/2018/. Accessed: 2019-05-27.

[30] StackOverflow Developer Survey Results 2019. https://insights.stackoverflow.com/survey/2019/. Accessed: 2019-05-27.

[31] strace: linux syscall tracer. https://strace.io/. Accessed: 2019-05-28.

[32] strip. https://linux.die.net/man/1/strip. Accessed: 2019-10-13.

[33] The Cargo Book. https://doc.rust-lang.org/cargo/. Accessed: 2019-05-24.

[34] The Clean Code Blog: The Single Responsibility Principle. https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. Accessed: 2019-09-19.

[35] The Development of the C Language. http://www.bell-labs.com/usr/dmr/www/chist.html. Accessed: 2019-06-21.

[36] The Go Programming Language. https://golang.org/. Accessed: 2019-04-24.

[37] The Rust Programming Language. https://doc.rust-lang.org/stable/book/. Accessed: 2019-04-25.

[38] The Rust Programming Language - Bibliography. https://doc.rust-lang.org/1.30.0/book/first-edition/bibliography.html. Accessed: 2019-04-29.

[39] The Rust Standard Library. https://doc.rust-lang.org/std/index.html. Accessed: 2019-09-19.

[40] The Traditional Vi. http://ex-vi.sourceforge.net/. Accessed: 2019-05-28.

[41] time. https://linux.die.net/man/1/time. Accessed: 2019-10-13.

[42] top. https://linux.die.net/man/1/top. Accessed: 2019-10-13.

[43] Valgrind. http://valgrind.org/. Accessed: 2019-05-24.

[44] Vim - the ubiquitous text editor. https://www.vim.org/. Accessed: 2019-05-28.

[45] Weenix. https://cs.brown.edu/courses/cs167/assignments/weenix.html. Accessed: 2019-04-29.

[46] Pamela Bhattacharya and Iulian Neamtiu. Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 171–180. ACM, 2011.

[47] Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[48] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2012.

[49] B. Kitchenham and S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering, 2007.

[50] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[51] Jyrki Kontio, Johanna Bragge, and Laura Lehtola. The focus group method as an empirical tool in software engineering. In *Guide to advanced empirical software engineering*, pages 93–116. Springer, 2008.

[52] Alex Light. Reenix: Implementing a Unix-Like Operating System in Rust. Honors thesis, Brown University, 2015.

[53] Leo A Meyerovich and Ariel S Rabkin. Empirical Analysis of Programming Language Adoption. In *ACM SIGPLAN Notices*, volume 48, pages 1–18. ACM, 2013.

[54] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[55] Karl Rikte. Using Rust as a Complement to C for Embedded Systems Software Development (A Study Performed Porting a Linux Daemon). Master's thesis, Lund University, 2018.

[56] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[57] Bjarne Stroustrup. Bjarne Stroustrup's C++ Style and Technique FAQ. `http://www.stroustrup.com/bs_faq2.html#finally`. Accessed: 2019-03-07.

[58] Florian Wilkens. Evaluation of performance and productivity metrics of potential programming languages in the HPC environment. Bachelor thesis, University of Hamburg, 2015.

Appendices

Appendix A
Interview Guide

Gudelines

- Personal experiences are preferred, but any insights are appreciated.
- Try to answer the questions as you interpret them, but feel free to ask for clarification if needed.

Questions

Q1 Please tell us a little about what you do here at Axis.

Q1.1 How long have you worked at Axis? At this department?

Q1.2 What is your current position at Axis?

Q1.3 What would you say are typical tasks that you work with?

Q2 Which programming languages have you worked with?

Q2.1 Which programming languages do you work with now? What do you think about them?

Q2.2 What performance aspects are important for your work? Most important? Hard requirements?

Note: Factors include Execution time, Parallelism, Compilation time, Memory usage, and Size of executable.

Q2.3 Do you have a favourite programming language? Which one and why?

Q2.4 What language features do you like/dislike and how do they affect your work/productivity?

Note: E.g. syntax, typing, functions.

Q3 What are your thoughts about C/C++?

Q3.1 How long have you worked with C/C++?

Q3.2 What are your feelings towards C/C++? Pros/Cons?

Q3.3 Is there anything that you feel C/C++ is missing?

Q3.4 What are your thoughts on the readability of C/C++ code?

Q4 What are your thoughts about Rust?

Q4.1 What do you know about Rust?

Q4.2 What are your feelings towards Rust?

Q5 Have you worked in projects where parts of the code were written in different languages?

- Q5.1 How did the usage of multiple languages affect development?
- Q5.2 What were the main benefits/disadvantages?
- Q5.3 What challenges did you face?

Q6 What are your experiences with code defects at the Streaming department?

- Q6.1 What types of code defects do you encounter?

  Note: Types of defects include Algorithm, Concurrency, Memory, Programming, Security, Performance, Failure, and Other.

- Q6.2 Do you encounter some types of code defects more often than other types? Which ones?
- Q6.3 Are some types of code defects harder to solve than others? Which ones?

Q7 What tools do you use?

- Q7.1 What IDE(s) do you use? Why?
- Q7.2 Do you use any tools for debugging/testing?
- Q7.3 Does your work impose any special requirements on the compiler? What are they?
- Q7.4 What tools do you use for collaboration?

Q8 Where do you look for answers to programming-related questions?

Note: E.g. StackOverflow, Reddit, documentation, co-workers.

Q9 How is your work affected by Open Source Software?

- Q9.1 Do you use Open Source Software in your work? Which ones?
- Q9.2 Does your work involve contributing to Open Source Software? How are these contributions made?

Q10 How do you feel about a transition from C/C++ to Rust?

- Q10.1 What challenges do you see in a transition from C/C++ to Rust?
- Q10.2 How do you think a transition to Rust in Streaming would affect other departments?

Appendix B
Focus Group Guide

Execution level

- What do you think about these factors?
- Is there any factor you would like to add to this level?
- What changes would you like to see, and what changes would you not like to see, due to a transition?

Programmer level

- What do you think about these factors?
- Is there any factor you would like to add to this level?
- What changes would you like to see, and what changes would you not like to see, due to a transition?

Toolchain level

- What do you think about these factors?
- Is there any factor you would like to add to this level?
- What changes would you like to see, and what changes would you not like to see, due to a transition?

Management level

- What do you think about these factors?
- Is there any factor you would like to add to this level?
- What changes would you like to see, and what changes would you not like to see, due to a transition?

Finally

- What do you think is important to keep in mind when transitioning to another programming language?
- What do you think would be the first steps, for Streaming, when doing a transition?
- Is there anything you would like to add?

Appendix C

Summary of the final Programming Language Transition Framework

Execution level		
E1	Execution time	How the execution time is affected.
E2	Parallelism	How the speedup gained from parallelisation is affected.
E3	Compilation time	How the compilation time is affected.
E4	Memory usage	How the amount of memory used during execution is affected.
E5	Size of executable	How the size of the compiled program is affected.
E6	Compatibility	How suitable the language is for specific tasks and hardware.
E7	*Memory management*	*How memory is used by the language.*
Programmer level		
P1	Readability	How easy it is to understand code.
P2	Writeability	How easy it is to write code.
P3	Learnability	How easy it is to learn the language.
P4	Knowledge base	How much knowledge is available about the language.
P5	*Debuggability*	*How easy it is to debug code.*
P6	*Testability*	*How easy it is to test code.*
Toolchain level		
T1	Testing	How the usage of testing tools is affected.
T2	Debugging	How the usage of debugging tools is affected.
T3	Co-operation	How the usage of co-operation tools is affected.
T4	*Build system*	How the usage of *build systems* is affected.
T5	Development	How the usage of development tools (e.g. IDEs) is affected.
Management level		
M1	Integration	How easy it is to integrate software written in the language.
M2	OSS collaboration	How OSS collaboration is affected.
M3	Productivity	How productivity is affected.
M4	Code defects	How work related to code defects is affected.
M5	*Education*	*Availability of education in the programming language.*
M6	*Maintainability*	*How work related to maintenance is affected.*
M7	*Recruitment*	*How recruitment is affected.*

Table C.1: A summary of the factors in the Programming Language Transition Framework, including changes suggested by the focus group. The changes are in italic.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-10-31

EXAMENSARBETE Transitioning from C to Rust in Media Streaming Development
An Industrial Case Study
STUDENTER Samuel Johansson, Ludvig Rappe
HANDLEDARE Elizabeth Bjarnason (LTH), Jonathan Karlsson (Axis), Srimanta Panda (Axis)
EXAMINATOR Björn Regnell (LTH)

From C to Rust in Media Streaming

POPULÄRVETENSKAPLIG SAMMANFATTNING **Samuel Johansson, Ludvig Rappe**

We compared C with the relatively new language Rust, using a Programming Language Transition Framework. We found that they perform similarly, but Rust is a clear winner in terms of usability.

For a long time, the C language has been very popular in performance-critical applications due to the level of control and performance it provides. While other languages like Java and C# continuously receive updates that improve developer experience, C has remained mostly unchanged in this respect. Some will probably argue that this is the beauty of C – that it is simple and not bloated with features upon features. But what do you do if you want the usability features from modern programming languages in addition to the control and performance provided by C?

Rust is a programming language released in 2015 that aims to provide safety without sacrificing performance. It brings a lot of conveniences for a low-level language, with features inspired by object-oriented and functional programming. The Rust language has been voted the most loved programming language in the Stack Overflow developer survey each year since its release, and is being used by companies such as Dropbox and Mozilla.

Axis Communications AB – leader in network cameras – have become interested in this new language and are now looking into the possibilities of doing a transition from C to Rust. We were invited to investigate the impact of such a transition by comparing C and Rust in the domain of media streaming. We did this guided by a Programming Language Transition Framework (PLTF) that we created. The PLTF is a list of factors involved in software development that could be affected by going from one language to another.

During our time of using Rust, we found it to be excellent to work with. The strict compiler stands out as one of the highlights, finding tons of errors during compile-time and even suggesting how to fix them. A lot of the time we could simply rely on the compiler telling us what to do without having to look in the documentation! However, we also found that it took quite some time to learn Rust – especially its ownership rules, which force you to think a lot more about how you write code than you need to do in other languages.

We obtained mixed results from comparing the performance of C and Rust. C performed best for one program, having a 20.44% shorter execution time and using 1.63% less memory, compared to the Rust version. For the other program however, we instead found that Rust had a 23.24% shorter execution time and used a surprising 30.41% less memory, compared to the C version.

Our results show no clear winner in terms of performance, but they do show that Rust can perform on par with C while providing a far superior developer experience. We also found that our PLTF helped us compare C and Rust in a structured way and hope that it will serve as a useful starting point for others looking to compare programming languages.