Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Writing Network Drivers in Rust

Simon Ellmann

Abstract

Many developers consider writing network drivers an unpleasant task. There are mainly three reasons for this aversion: The difficulties of working in kernel space, the complexity of most drivers and a general reluctance to C programming.

Fortunately, there are alternatives nowadays. Kernel modules no longer necessarily have to be written in C, an increasing number of hardware offloading features in modern network cards allows for far less complex drivers and the upswing of user space network drivers obviates the need for writing kernel code altogether.

We show that driver development can be challenging but rewarding by presenting a state-of-the-art user space network driver written in Rust, designed for simplicity, safety and performance. With 1,306 lines of code in total and less than 10% unsafe code, the driver focuses on the bare essentials of packet processing while still beating the kernel and several other user space drivers with a forwarding capability of more than 26 million packets per second on a single 3.3 GHz CPU core.

We discuss our implementation and evaluate it from different points of view. From our results, we conclude whether Rust is a good programming language for writing network drivers.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In recent years more and more developers are moving drivers from kernel space to user space. One of the main reasons for this development in relation to network drivers is the performance bottleneck of the socket API. The general-purpose kernel stack is just too slow for modern requirements. In the past, developers were writing their own kernel drivers to circumvent this problem. But driver development in the kernel is a cumbersome process that requires painful accuracy since programming mistakes at such low level can and will crash the kernel eventually. Besides, the kernel imposes various restrictions on the development environment and available tools inside the kernel.

Thankfully, there are alternatives to driver development in the kernel. In Chapter 2 we illustrate these alternatives by showing different approaches to low-level network communication in Linux. We also talk about the advantages of user space drivers. One of the most pleasant benefits regarding them is the ability to choose any programming language for the implementation. However, most user space drivers are still written in C, an ancient programming language that leads to buffer and stack overflows, segmentation faults, memory leaks and other undefined behavior if not handled carefully. Because of the ability to use any programming language, the question arises which programming language is particular suitable for network drivers.

To answer this question, we discuss desirable properties of programming languages for network drivers in Chapter 3 and suggest – as an alternative to C – Rust for network programming, a state-of-the-art programming language that promises to fulfill all our favored properties. We talk about Rust in detail by presenting its core features and some Rust-unique concepts.

To prove Rust's suitability for network drivers we reimplemented the ixy driver, a fast and lightweight user space network driver written for educational purposes by Paul Emmerich, Maximilian Pudelko, Simon Bauer and Georg Carle. We lay out the driver's main characteristics in a few sentences in Chapter 4 and explain why ixy is particularly suitable for our task.

We then present our implementation in Chapter 5 in much detail. We show which data structures and algorithms are used inside the driver and how they work together. In Chapter 6 we evaluate the performance and other aspects of our code.

In Chapter 8 we draw our conclusions about using Rust as a programming language for network drivers while Chapter 7 presents Redox, an operating system written entirely in Rust that contains two real-world network drivers.

Chapter 2

Network Communication in Linux

Although the Internet is ubiquitous nowadays, network communication, network cards and network drivers in particular are still often seen as black boxes by developers and users. For users, most of the time it is sufficient to plug an ethernet cable into their computer to go online. For developers, high-level APIs provided by the operating system let them include network communication features in their programs easily while not having to deal with the intricacies of low-level network programming.

However, if there is no driver for a certain network card yet or when the performance of packet processing becomes relevant, it is necessary to gain rudimentary knowledge of the foundations of network communication, especially of the way applications interact with the operating system and the network card. Every time data is sent from an application, the data is processed by several layers until it arrives at the network card. A good model – the de facto standard – to characterize these layers is the OSI model

7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data link
1	Physical

Figure 2.1: Communication layers of the OSI model.

shown in Figure 2.1. Every layer in the model provides different services like packet segmentation (layer 4), routing (layer 3) or reliable transmission (layer 2) to the layers above.

Usually, the application is responsible for application, presentation and session layer, while transport and network layer are handled by the operating system through the general-purpose kernel network stack. The lowest two layers, the physical layer and the data link layer are controlled by the network interface card (NIC) and the network driver.

Traditionally, the different tasks of network application and network driver have been separated in Linux by user space and kernel space due to Linux's operating system design. Nevertheless, there are different approaches to low-level packet processing in Linux: Most applications use the socket API of the kernel, while some applications run their own kernel modules, some handle everything in kernel space and some handle everything in user space.

2.1 Kernel Space

Linux is based on a monolithic kernel, which means the whole operating system is working in kernel space while all other services operate in user space. The kernel provides the socket API to user space applications, i.e. various functions and data types for a simple-to-use interface on so called *Berkeley* or *POSIX* sockets. These sockets are special files that can be used to communicate with local processes or distant hosts by plain read and write operations, following the Unix concept of "everything is a file". A user space application that wants to establish a communication channel issues a call to the `socket()` function that eventually creates an instance of a socket struct in the kernel and returns a file descriptor for that socket to the application. Subsequently, `bind()` and `connect()` or `listen()` and `accept()` associate a connection between two or more processes to a socket and `send()` and `recv()` are used for sending and receiving data to or from a socket.

Using sockets is a convenient way for applications to communicate with other hosts since the API provides a simple yet powerful interface for inter-process communication and the kernel manages the network card. However, using the socket API is slow, since the kernel allocates huge structs with countless metadata fields for every packet and due to the fact that all packets have to be copied from kernel space to user space and back. This is especially noticeable if the kernel has to handle a vast number of small packets. To diminish the problems of the general-purpose kernel stack, some applications implement their own kernel modules. Two well-known examples for this are Open vSwitch [13] and the Click Modular Router [10].

Nevertheless, using a custom kernel module might still not be fast enough due to the frequent context switches between kernel and user space. This can be mitigated by operating entirely in kernel space. Although applications in kernel space allow for very fast packet processing, writing them is cumbersome because of missing debugging tools in the kernel, general restrictions like no floating-point operations and the ubiquitous thread of crashing the system in case of bugs in the software.

2.2 User Space

Another approach to fast packet processing are applications working entirely in user space (including their own network stack/driver) like DPDK [2] and Snabb [5]. Special device files that can be mapped into memory with root privileges allow applications to access network cards and other physical devices from user space.

Although some operations like interrupt handling can be tricky, user space drivers mitigate many problems concerning kernel space drivers. They are generally simpler and more flexible, can be written in any programming language and profit of all the tools that are normally available for software development. Bugs have less severe consequences, and performance can be significantly better since there are far less context switches and received and sent packets do not necessarily have to be copied between user and kernel space. DPDK, for example, is able to operate at a performance very close to line rate.

Because of these advantages we chose to develop our driver in user space, although it is possible to write kernel modules in Rust as well [8].

Chapter 3

Choosing Rust for Drivers

Programming languages are the basic tools of any developer. Thousands of them have been created over the years. One of the first high-level programming languages was Plankalkül, created by Konrad Zuse between 1942 and 1945, followed by FORTRAN, COBOL and many others up to C, C++ in the seventies/eighties and modern languages like Kotlin and Swift. The sheer mass of programming languages alone raises the question which programming languages are generally and which ones are particularly suitable for network drivers.

Every language follows various concepts also known as programming paradigms that classify how code is organized and how it is executed. Since programming languages differ fundamentally, it is necessary to choose an appropriate programming language that suits our task of writing network drivers. Generally speaking, network drivers are expected to be highly performant, reliable, i.e. without any undefined behavior and software bugs, simplistic, easy to use and comprehensible.

These requirements lead to some crucial design decisions regarding the programming language. First of all, to meet a high performance the language should be a compiled language instead of an interpreted language. Though interpreted languages are generally easier to implement and can be executed "on the fly", they have to be interpreted (thus the name) while being executed and lack the opportunity for powerful optimizations while compiling. Therefore, languages like Python are slower and should not be our first choice.

Regarding reliability and safety, the compiler should reject any unsafe code and alert us on programming errors as far as possible. Although the C/C++ compilers are improving continuously, there are still far too many situations where they do not foreclose undefined behavior by warning the developer.

Another point to be considered for the development phase is the ecosystem of a programming language. It consists mainly of the tools available for programming like compilers, development environments, bug checkers, etc., documentation and usually a developer or user community.

Summarized one can say that the perfect programming language for a network driver would be a highly performant compiled language that is memory safe, simple to read and use and that provides great documentation and other development aids. Fortunately for us, there is a relatively new language that claims to fulfill all of these goals: Rust.

Rust is a new systems programming language developed by Mozilla and others to provide a "safe, concurrent and practical" programming language. It was publicly announced in 2010 by Mozilla, the first stable release, Rust 1.0, was published on May 15, 2015. Rust is syntactically similar to C++ but intends to offer better memory safety und trustworthy concurrency while being highly efficient through zero-cost abstractions. It is a compiled programming language that combines different paradigms like concurrent, functional and generic programming. Rust consists of a unique system of ownership, moves and borrows that is enforced at compile time and makes garbage collection unnecessary. This system is the key to meeting Rust's goal of memory safety [1, 9].

3.1 Syntax

Rust's syntax is closely related to the syntax of C and C++. Statements are separated by a semicolon, code blocks enclosed by curly brackets. The control flow is ruled by keywords such as `if`, `else` and `while`. Nevertheless, differences like `for` that introduces for-each-loops in Rust, `match` for the more powerful pattern matching instead of `switch` and a missing ternary conditional exist. Functions are declared with the `fn` keyword and variables with `let`. Functions and variables are generally written in snake case while data types use a camel case style. In addition, Rust supports functional programming like lambda functions that have their own syntax.

3.2 Type System

Rust is a statically typed programming language. Although the Rust compiler must know the types of all variables at compile time, that does not imply that these types have to be declared explicitly. Rust features type inference to determine the types of variables. Failing assignments of values to variables in the code due to conflicting or unknown types lead to compile time errors. To assign values multiple times to a variable, the variable has to be declared with the `mut` keyword as variables in Rust are by default immutable.

User defined data types can be declared as `structs` or `enums` (tagged unions). Using the `impl` keyword, methods can be declared on these user defined data types (like it is possible with classes in other languages). A mechanism similar to type classes is available in Rust, called "traits". Traits are a set of methods that can extend the functionality of a class. They allow for ad hoc polymorphism by adding constraints to type variable declarations and are inspired by Haskell. For instance, implementing the `Add` trait allows to use the `+` operator with user defined types. Functions can be called with generic parameters that are usually required to implement one or multiple traits. Instead of inheritance, Rust uses composition by combining traits.

3.3 Memory Management and Safety

Rust's most essential feature is its unique ownership system. Ownership allows Rust to make memory safety guarantees while avoiding garbage collection. Although plenty of languages make use of a garbage collector that automatically keeps track of objects and frees them when not being used anymore, this is not without two major drawbacks.

First of all, cleaning up resources with a garbage collector is non-deterministic. Garbage collectors are complex pieces of software and understanding why memory is not being freed can be a challenge. If a developer wants to ensure that a resource is cleaned up at a certain point, this typically means he either has to force the garbage collector to cleanup everything or wait until the resource is ready to be freed. Both of these options take away control from the developer and are disappointing.

Second, garbage collection leads to potential performance issues as the garbage collector needs a varying amount of CPU cycles to perform the cleanup. The time spent on garbage collection is especially high if there is a lot to be cleaned up. This can be problematic on real-time applications like online games or – as in our case – network drivers.

Without a garbage collector, the developer is responsible to return memory to the operating system when it is no longer used. The amount of security problems listed in public databases concerning memory misuse proves that this is a difficult responsibility to meet. If a developer forgets to free memory, he is wasting memory and the application will crash when it is running out of memory. If a developer frees memory too early, there will be an invalid reference also known as a dangling pointer, which is undefined behavior as is freeing memory twice.

C and C++ are well known for these issues. Rust does not accept these drawbacks and takes a different approach by restricting the way pointers are used through its ownership system. Rust's rules enable its compiler to verify at compile-time that a program is free of memory safety errors, i.e. the before mentioned dangling pointers, double frees, etc. Yet it aims to keep the developer in charge of the memory while preventing unsafe operations. At runtime there is no difference to programs written in C/C++ except that the compiler has proven that memory is handled safely. In fact, the same rules are the basis for safe concurrent programming as they prevent data races and data corruption. Having a thread-safe programming language is a valuable point since most network cards use multiple receive and transmit queues and can therefore be managed by different threads.

3.4 Ownership

Rust's ownership system consists of three simple rules. In Rust, every value has an owner. There can only be one owner at a time for a certain value. When this owner goes out of the scope, i.e. its lifetime ends, the value will be freed. In Rust terminology this is called "dropping" a value and is a pattern similar to *Resource Acquisition Is Initialization* (RAII) from C++.

The scope of a variable is the range within a program for which the variable is valid, e.g. variables defined inside a function become invalid when the function returns, and their values are freed. Ownership can be transferred from one variable to another. This is called "moving" a value and happens when a function is called with a parameter, a value is assigned to a variable, or a function returns a value. This value is then not copied but moved.

To be precise, that is not true for all values. Types with a known size at compile time like integers, booleans, floats, etc., are copied as they are stored entirely on the stack and it is thus cheap to copy them.

When a value has been moved, the previous reference becomes invalid and cannot be accessed anymore to ensure memory safety, i.e. no dangling pointers. To pass a value to a function without transferring ownership, i.e. moving the value, Rust uses references.

A reference to a value can be created with the `&` operator. Having references as function parameters is called “borrowing” since the value cannot be used by other functions if a mutable reference has been passed until the borrowing function returns.

Similar to variables, there are mutable and immutable references, and all references are immutable by default. In relation to references, Rust enforces additional restrictions: There can be unlimited immutable references to a variable or a single mutable reference, but not both. A value has to live at least as long as any reference to that value. This means we cannot return a reference to a value that was declared inside a function.

Besides, a value cannot be moved as long as there is a reference to that value. Listing 3.1 and Figure 3.1 illustrate why this is forbidden in Rust. `s` is a variable that points to the string “hello” on the heap. It contains the length of the string as well as the capacity of the memory buffer and the actual pointer and is stored on the stack. `r` is a reference to `s`. Up to this point, i.e. line 3 in Listing 3.1, the code is perfectly legit. However, the assignment of `s` to `t` moves `s`, leaving the previously used memory empty and turning `r` into a dangling pointer. To eliminate this error, Rust disallows values to be moved while being referenced, and they cannot be modified either to prevent data races.

```rust
1 fn main() {
2     let s = String::from("hello");
3     let r = &s;
4     let t = s;
5 }
```

Listing 3.1: Attempt to move a borrowed value.

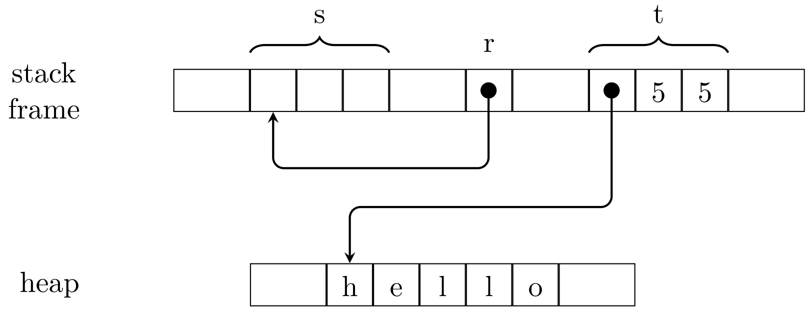


Figure 3.1: Reference to a string that has been moved away. Based on “Programming Rust: Fast, Safe, Systems Development” [1].

3.5 Unsafe Code

The ownership system of Rust is very powerful. However, static analysis is quite conservative and still subject to limited decision capabilities. There are valid programs that are rejected by the compiler when the compiler is unable to determine whether or not the code upholds the required guarantees.

To make the Rust compiler accept this kind of code, Rust uses the keyword `unsafe`. Using the unsafe feature is similar to signing a contract: the developer is now responsible to follow the rules to avoid undefined behavior as the compiler is unable to enforce them automatically.

With unsafe code it is possible to use four additional features of Rust. This includes dereferencing raw pointers, calling unsafe functions, including functions from Rust's foreign function interface, accessing and modifying mutable static variables and implementing unsafe traits.

When using these features, the developer has to take care that he follows the rules. For example, dereferencing a pointer beyond the end of its original referent is forbidden. It breaks the contract of using `unsafe` and leads to undefined behavior. Lots of features in Rust have rules to follow, but as long as the possible consequences of misusing them does not lead to undefined behavior, they do not require the `unsafe` keyword.

Therefore, having unsafe code inside of a function does not necessarily imply that that function has to be marked as unsafe. In fact, declaring a function as `unsafe` is independent of whether that function uses code that requires unsafe or not. There are rare cases – like `Packet::new()` in our driver – where the function itself does not include any unsafe code but should nevertheless be declared as unsafe.

Idiomatic Rust code often means that unsafe code is wrapped inside a safe API. A good example for a method that requires unsafe code is `split_at_mut()`, a function that takes a slice[1] and splits it into two slices at a given index. Listing 3.2 shows a naive implementation for `split_at_mut()`.

[1] a reference to a contiguous sequence of elements in a collection similar to arrays in other languages but with an unknown size at compile time

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

Listing 3.2: Naive implementation of `split_at_mut()`, taken from “The Rust Programming Language” [9].

The function asserts that the index at which the slice shall be split is within the slice. If the assertion fails, the program terminates immediately. In Rust, terminating due to an unrecoverable programming error is called “panicking”.

Unfortunately, Rust’s borrow checker is unable to understand that the function borrows different parts of the slice which is valid to do and returns an error when compiling, stating that there are two mutable references to the same slice at the same time. Using unsafe Rust code, it is possible to implement `split_at_mut()`. Listing 3.3 shows the generic implementation from the Rust standard library, slightly modified for readability here.

```
pub fn split_at_mut(slice: &mut [T], mid: usize) -> (&mut [T], &mut [T]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    unsafe {
        assert!(mid <= len);

        (from_raw_parts_mut(ptr, mid),
         from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

Listing 3.3: Actual implementation of `split_at_mut()` including unsafe code.

Just like the naive implementation, the function asserts that the index is inside the slice and either panics or returns two mutable references to the different parts by calling `from_raw_parts_mut()` with a raw pointer to each part of the slice and the lengths of the new slices.

`from_raw_parts_mut()` is unsafe because the user has to ensure that the raw pointer is valid and indeed points to a slice (with the correct length). The `offset` method is unsafe because it cannot be sure that the pointer at the given offset is valid. However, the combination of the two unsafe functions together with the previous assertion is a safe abstraction and therefore appropriate use of unsafe code.

By forcing developers to use `unsafe` for code possibly leading to undefined behavior, Rust makes sure that developers are aware of their actions, i.e. do not use `unsafe` features unknowingly, and hopefully refrain from unsafe code as far as possible which is desirable for any computer program and especially for network drivers.

Chapter 4

Ixy

Ixy is a network driver written to show how network cards work at the driver level. It is implemented entirely in user space with an architecture similar to DPDK and Snabb.

The primary design goals of the ixy network driver written in C are simplicity, no dependencies, usability and speed. While Snabb has very similar design goals, the ixy C version tries to be “one order of magnitude simpler”. Thus, a simple forwarder and the driver consist of less than 1000 lines of C code. As there are no external libraries and no kernel code, the different code-levels can be explored within a few steps. Every function is only a few calls away from the application logic. Some features (like various hardware offloading possibilities) have been left out to keep the driver as simple as possible.

Initialization and operation of the driver is very similar to Snabb while memory management, batching and abstraction come from DPDK. Ixy provides two different implementations, ixgbe and VirtIO. On initialization the appropriate driver is selected and a struct containing function pointers to the driver-specific implementation is returned to the user. Ixy offers functionality to initialize a network card at a given pci address and send and receive packets. The framework also exposes device statistics to conduct performance measurements as well as a safe packet API with custom-built data structures to allocate and free packets in its memory pools. Applications include ixy directly, and using these abstractions leads to a comfortable handling of the driver. Users can read it from top to bottom without any complex hierarchies.

Ixy uses NICs of the ixgbe family because these cards are very common in general purpose servers and Intel releases extensive datasheets about them.

As ixy tries to “take the magic out of user space network drivers” with its simplicity, usability and speed, it is very well-suited as a template for implementing these kinds of drivers. Because of this and to be able to compare our work with future implementations we used ixy as a reference implementation for our user space driver written in idiomatic Rust code.

Chapter 5

Implementation

The implementation of the ixgbe driver relies heavily on the original ixy C driver and the Intel datasheet. Any code references in the following sections refer to commit `e193471` of the `master` branch of our implementation where not stated otherwise. Page numbers and section numbers regarding the Intel datasheet refer to revision 3.3 (March 2016) of the 82599ES datasheet [6].

5.1 Design

Ixy aims to be simple, fast and usable. All the same applies to our implementation. But we add another goal to our project: safety.

Rust by itself is a safe programming language as long as developers avoid the `unsafe` keyword. Thus, our implementation of the driver tries to minimize the use of unsafe code where this is reasonable. We will explain where and why there is unsafe code in the implementation. Currently, the driver uses about one hundred lines of unsafe code in total.

Unlike ixy in C, we do not focus on no dependencies. With `crates.io` and `cargo`, Rust provides a great package management system and actively encourages developers to use packages provided by others, following the Unix philosophy of making each program do one thing well. As with unsafe code, we will state where and why we included packages (called "crates" in Rust). So far there are four crates in our dependencies. One of them is the `log` crate to provide a logging API so that the users of our driver can decide whether they want to use logging and if so in which form.

We provide two sample applications with our driver to show how the driver can be used. One application is a packet forwarder and the other one is a packet generator. The two applications show how the logging API works by writing all output of the driver to `stdout`.

5.2 Architecture

The architecture of our implementation is based on the architecture of the C driver. To provide a simple interface to the users of our driver, we use a trait that is implemented by a greatly reduced version of the Intel ixgbe driver and could by implemented by a VirtIO version in the future as well (similar to ixy). That trait is the public interface for all applications employing our driver. It offers methods to initialize an ixgbe network card at a given pci address and send and receive packets with it. Using a trait is much more elegant than the simple abstraction provided by the C implementation. We implement all driver related functions as methods on our device while in C a pointer to the device has to be passed to every function. Just like the original ixy driver, we expose device statistics for performance measurements and provide a safe API for all memory operations.

5.3 Security Considerations

It is important to note that our implementation requires root privileges to access the PCIe device. However, a custom-built kernel module would still be worse as it has – because of its nature – root privileges and has to be written in C, an unsafe programming language.

In theory, the driver could drop its privileges after initializing the device with `seccomp(2)`. Unfortunately, this is not enough since the device is still under full control of the driver and can write to any memory regions through direct memory access (DMA).

To write a truly secure user space network one would have to make use of the I/O memory management unit (IOMMU), a modern virtualization feature to pass PCIe devices into virtual machines. In Linux this can be done using `vfio`, a framework specifically designed for “safe, non-privileged, userspace drivers” [11]. Making use of `vfio` should be considered for future work on the driver as it goes beyond the scope of this thesis.

5.4 Initialization

Preparing a NIC of the 82599ES family to send and receive packets requires a single call to the `ixy_init` function with three parameters containing the PCIe address of the network card and the desired number of send and receive queues.

To communicate with and manage the PCIe device, Linux offers a pseudo filesystem called `sysfs`. The driver reads from and writes to special files to change the state of the device. For instance, to unbind any kernel driver using the network card before device initialization, the driver writes the NIC's PCIe address to the `unbind` file. To enable DMA, two bytes are read from the `config` file, modified and written back. Previous to these actions, we do a rudimentary check if the PCIe device is indeed a network card by reading the device class from the `config` file. We include the `byteorder` crate in our driver to be able to read different sized integers (with the correct endianness) from this file instead of having to fiddle with raw bytes and unsafe code.

Following unbinding and DMA-enabling, the file exporting the base address registers (BARs) of the device is mapped into memory as shared memory which means that changes to the memory mapped region, i.e. the different registers, are written back to the file and vice versa. The base address registers are used to configure the network card. The offsets of all registers are documented in the Intel datasheet. We use a slightly reduced copy of `ixgbe_type.h` (containing all offsets as `#define`s and some structs) from the ixy driver that we translated into Rust code by annotating structs as C structs, replacing the `#define`s by Rust's equivalent `const`, rewriting all ternary operators and so an. Technically speaking, using this constants file increases our code base by more than 4,000 lines of code. However, we use less than one hundred out of these 4,000 lines and having a file with all registers on hand is very comfortable for future work on the driver.

For the memory mapping, we do not use a special crate from `crates.io` since working with raw pointers is more comfortable with our offset constants and offset functions and implementing a safe API on top of the raw pointers is straightforward.

Listing 5.1 shows how we memory map files in our driver. The mapping function opens a file at a given path with read-write access and passes the file descriptor and the length of the file to `libc::mmap()` that maps the file as shared memory into memory and returns a pointer to the memory location. `libc::mmap()` has to be inside an `unsafe` block as it is just a call to `mmap(2)` from the C standard library (a foreign function). If `mmap(2)` returns a null pointer or the length of the given file equals zero, an error is returned from the function. Otherwise the pointer and the length of the mapped file are returned.

```
pub fn mmap_file(path: &str) -> Result<(*mut u8, usize), Box<Error>> {
    let file = fs::OpenOptions::new().read(true).write(true).open(&path)?;
    let len = fs::metadata(&path)?.len() as usize;

    let ptr = unsafe {
        libc::mmap(
            ptr::null_mut(),
            len,
            libc::PROT_READ | libc::PROT_WRITE,
            libc::MAP_SHARED,
            file.as_raw_fd(),
            0,
        ) as *mut u8
    };

    if ptr.is_null() || len == 0 {
        Err("mapping␣failed".into())
    } else {
        Ok((ptr, len))
    }
}
```

Listing 5.1: Memory-mapping in Rust.

To call foreign functions from Rust we have to specify the function's signature and annotate it with `extern`. This is called a binding. The `libc` crate provides bindings for all functions from the libc, so we use that crate in our driver to not have to worry about bindings.

After memory-mapping the device file, interrupts are disabled to prevent re-entrance as recommended in the datasheet, the device is reset, interrupts are disabled again since we do not support interrupt handling, auto negotiation for the network is enabled and the statistical counters of the device for received and sent packets are reset.

```
fn set_reg32(&self, reg: u32, val: u32) {
    assert!(
        reg as usize <= self.len - 4 as usize,
        "memory␣access␣out␣of␣bounds"
    );

    unsafe {
        ptr::write_volatile((self.addr as usize + reg as usize) as *mut u32, val);
    }
}
```

Listing 5.2: Setting a 32-bit register in Rust.

```c
1 static inline void set_reg32(uint8_t* addr, int reg, uint32_t value) {
2     __asm__ volatile ("" : : : "memory");
3     *((volatile uint32_t*) (addr + reg)) = value;
4 }
```

Listing 5.3: Setting a 32-bit register in C.

All of these actions happen by getting and setting some registers of the mapped device. Listing 5.2 and Listing 5.3 show the differences between the `set_reg32()` function in Rust and in C.

The C implementation uses a compiler barrier in line 2 to prevent the compiler from resorting the following memory access to `addr + reg` and the `volatile` keyword in line 3 is there to prevent the compiler from optimizing away the memory access altogether. Rust misses this compiler barrier since it is unnecessary for x86 architectures. The same effect as `volatile` is achieved by using `write_volatile()` from the `ptr` module. Since writing to an arbitrary memory location is unsafe, `write_volatile()` requires an `unsafe` block. By asserting that the register is indeed inside the mapped memory of the device, we provide a safe abstraction and do not have to mark `set_reg32()` as `unsafe`.

5.5 DMA

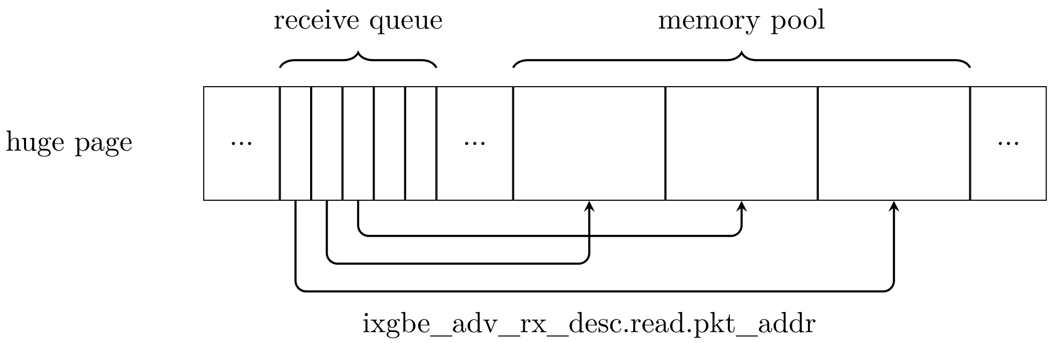


Figure 5.1: Receive queue with DMA descriptors pointing to packet buffers in the memory pool..

After resetting the device, the queues for receiving (RX) and transmitting (TX) packets are initialized. Figure 5.1 shows the general structure of the receive queues. Every queue is a ring buffer filled with descriptors containing pointers to the physical addresses of the packets as well as some metadata about the packets, e.g. their size. The network card can be configured to split traffic on multiple queues using filters or hashing. For a simple setup, using one receive and one transmit queue is sufficient.

To initialize the receive and transmit queues, memory has to be allocated for the queues. This memory has to stay resident in physical memory since the network card uses the memory independently of the central processing unit (CPU), i.e. accesses the memory via its physical addresses. In kernel space there is an API to allocate DMA memory, in user space we have to use other mechanisms since this API is not available here. To disable swapping, we can use `mlock(2)`. Unfortunately, this function is not part of Rust's standard library, but it is part of the libc. So, we use the binding from the `libc` crate and call `mlock(2)`. However, `mlock(2)` only ensures that the page is kept in memory. The kernel is still able to move pages to different physical addresses. To circumvent this problem, we use huge pages with a size of 2 MiB. These pages cannot be migrated yet by the Linux kernel, thus stay resident in physical memory.

Allocating DMA memory on huge pages is fairly simple. The `setup-hugetlbfs.sh` script in our repository creates a `hugetlbfs` mount point and writes the required number of huge pages to a `sysfs` file. Memory allocation is then accomplished by creating a new file in the mounted directory and mapping the file into memory using `mmap(2)` from the libc.

The physical address of the mapped memory can be derived via the `procfs` file `/proc/self/pagemap`. Every queue is matched to one huge page and the physical address is communicated to the network card through the BARs.

5.6 Memory Pools

As explained in the previous section, the descriptors of receive and transmit queues contain pointers to packet data which is read from and written to by the network card via DMA. Thus, the packet data has to be inside DMA memory, too.

Allocating memory every time a packet is received is a huge overhead. The general-purpose network stack in the kernel needs about 100 CPU cycles to perform these allocations. Using memory pools that manage already allocated memory decreases the amount of CPU cycles significantly.

Consequently, every receive queue has a memory pool attached to it for incoming packets. Transmit queues do not have their own memory pools as it suffices to pass packets from an existing pool to them.

```
struct mempool {
    void* base_addr;
    uint32_t buf_size;
    uint32_t num_entries;
    uint32_t free_stack_top;
    uint32_t free_stack[]; // contains the entry id
};
```

Listing 5.4: Memorypool in C.

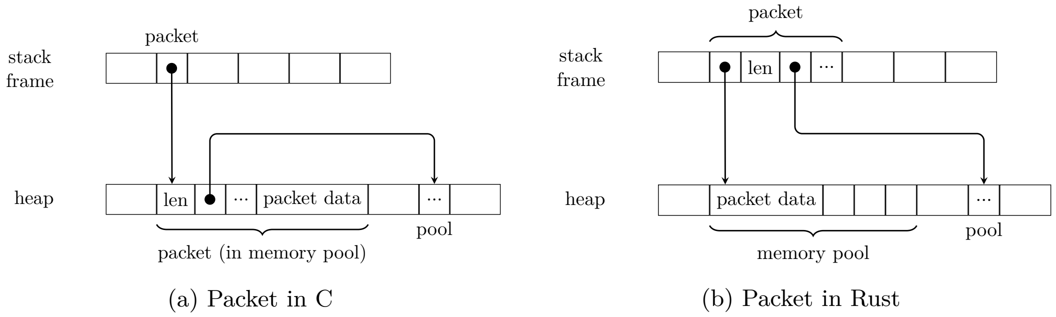
```
pub struct Mempool {
    base_addr: *mut u8,
    num_entries: usize,
    entry_size: usize,
    phys_addresses: Vec<usize>,
    pub(crate) free_stack: Vec<usize>,
}
```

Listing 5.5: Memorypool in Rust.

Listing 5.4 and Listing 5.5 show the structure of the memory pools in C and in Rust. In C, the memory pools are implemented as structs containing a stack of entry id's to free packet buffers. The virtual address of a buffer is calculated by `base_addr + (entry_id * entry_size)`. In Rust, it is all the same except that we use a `Vec<T>` (called Vector) for `free_stack` and another one to store the physical addresses of the buffers in the memory pool. Vectors are Rust's native data type for stacks, they are de facto contiguous growable arrays.

In C, the packet buffers from the memory pool are structs themselves with some header fields like the packet size or a reference to the memory pool and an unsized data field. In Rust it is possible to have structs with unsized fields, but they are not comfortable to handle and generally quite unusual. Besides, putting the packet header right next to the packet data that is written by the network card seems like an odd idea.

Hence we follow a different approach, displayed in Figure 5.2. The packet buffers do not contain any headers, just the packet data, and their physical addresses are stored inside a vector in the memory pool. There is still a stack inside the memory pool that keeps track of the packet buffers. Whenever a packet is returned to the user, a `Packet` struct is instantiated that contains all necessary information: a pointer to the packet data, physical address of the packet data, size of the packet, a reference to the memory pool and the position of the packet buffer inside the memory pool. This `Packet` struct is shown in Listing 5.6.



(a) Packet in C

(b) Packet in Rust

Figure 5.2: Packet layout in C and in Rust.

```
pub struct Packet {
    pub(crate) addr_virt: *mut u8,
    pub(crate) addr_phys: usize,
    pub(crate) len: usize,
    pub(crate) pool: Rc<RefCell<Mempool>>,
    pub(crate) pool_entry: usize,
}
```

Listing 5.6: Packet in Rust.

It stands out that the reference to the memory pool is not a normal reference like `&Mempool` but a `Rc<RefCell<Mempool>>`. There are various reasons for this. First of all, we have to be able to mutate the referenced memory pool through the packets (for example when a `Packet` is dropped). Since we have multiple packets referencing the same memory pool and Rust does not allow multiple `&mut Mempool` at compile time, we use a `RefCell<T>`, a mutable memory field that checks the borrow rules dynamically. The `RefCell<Mempool>` lets us borrow the memory pool mutably at runtime by calling `pool.borrow_mut()`. It panics if we try to borrow the same memory pool mutably twice. This pattern of being able to mutate data even when there are immutable references to it is called *Interior mutability* in Rust.

Second, we cannot have a single owner for the memory pool since we need it to live as long as it is used by the driver for sending or receiving and as long as there are any packets left referencing the pool (this might be longer than `IxgbeRxQueue` exists). To have multiple owners for the pool, we use `Rc<T>`, a reference counter for `RefCell<Mempool>`. The memory pool is freed when the number of references to it counted by `Rc<T>` drops to zero.

As a consequence of having `Rc<RefCell<Mempool>>` inside the packets, we have to use this construct for all variables referencing memory pools inside and outside of our driver.

This is the reason why `Mempool::allocate()` returns a `Rc<RefCell<Mempool>>` instead of `Mempool`.

```
 1 fn forward(buffer: &mut VecDeque<Packet>, rx_dev: &mut impl IxyDevice,
 2            rx_queue: u32, tx_dev: &mut impl IxyDevice, tx_queue: u32) {
 3     let num_rx = rx_dev.rx_batch(rx_queue, buffer, BATCH_SIZE);
 4 
 5     if num_rx > 0 {
 6         // touch all packets for a realistic workload
 7         for p in buffer.iter_mut() {
 8             p[48] += 1;
 9         }
10 
11         tx_dev.tx_batch(tx_queue, buffer);
12 
13         // drop packets if they haven't been sent out
14         buffer.drain(..);
15     }
16 }
```

Listing 5.7: forward() from the forwarder application of the ixy driver.

The `Packet` struct implements three traits from the Rust standard library: `Deref`, `DerefMut` and `Drop`. Listing 5.7 shows the use of these traits in the `forward()` function of the forwarder application. We implemented `Deref` and `DerefMut` so that the packet's data can be accessed and modified by treating `Packet` as a slice of bytes (`&[u8]`). Line 8 shows how the application reads the 49th byte of packet `p` and increases it by one. We implemented `Drop` so that the packet's buffer is returned to the memory pool when a packet goes out of scope. This happens in line 14 when all packets are removed from the buffer.

Dereferencing the raw pointer of the `Packet` struct is unsafe just like returning a slice with `slice::from_raw_parts()` for the `Deref` and `DerefMut` trait. However, since we ensure that no slice with a size greater than the corresponding buffer is returned and only one object at a time uses the buffer, we can provide a safe abstraction and implement these traits.

5.7 Receiving Packets

During initialization of the device the driver fills all descriptors of the receive queues with physical pointers to packet buffers. The receive and transmit descriptor queues are actually rings that are accessed by both the driver and the device. The driver controls the tail pointer of the ring while the network card controls the head pointer. When a packet is received, a `Packet` struct pointing to the corresponding buffer in the memory pool is instantiated, the physical address of a free buffer is stored in the receive descriptor and the ready flag of that descriptor is reset. Since the receive descriptors do not contain virtual addresses of the data buffers we have to keep track of which buffer belongs to which descriptor. This is done using a `Vec<usize>` of references to buffers where the index of a buffer equals the index of the descriptor in the descriptor ring. Thus, the vector acts as a copy of the ring referencing the corresponding buffers.

Receiving and transmitting packets is done in batches to improve performance. The user passes a `VecDeque<T>` (Rust's native type for queues) to the receive function so that the receive function can push the received packets onto that queue. This is different to the C implementation. In C, the user passes an array to the function and the function puts pointers to buffers of received packets into that array. In Rust, we push `Packet` structs onto a queue. We could also use references to packets, but by returning packets we can explicitly pass ownership of the packet's memory to the user. As long as the user holds a certain `Packet`, he is the single owner of that `Packet` and its memory. When the user returns the packet to the driver by either calling the send function or dropping it, the packet's memory is freed, i.e. it is returned to the memory pool by pushing a reference to the buffer onto the free stack of the memory pool. If the user had held only a reference to the packet, he would have to return that reference explicitly to the driver since there is no way to verify if a reference has been dropped.

5.8 Transmitting Packets

Transmitting packets works similarly to receiving packets but is more complicated as sending packets is asynchronous. When a packet is to be sent, the descriptor at the current index in the transmit descriptor ring is updated with the physical address of the packet data, i.e. of the buffer in the memory pool, and the data length, and the entry id of the buffer is stored inside a `VecDeque<usize>`. It is necessary for the transmit queues to remember which buffers are still in use because otherwise the buffers would be returned to the memory pool too early and could be reused before the actual data has been sent out, thus sending other data than intended.

Therefore, the transmit function consists of two parts: verifying which packets have been sent out and returning the corresponding buffers to the memory pool (this is called cleaning), and sending new packets and storing their buffers for cleaning. Cleaning the transmit queues is done in batches to reduce the amount of costly PCIe transfers.

Chapter 6

Evaluation

To evaluate the performance of our implementation, we run the forwarder application under a full bidirectional load of 29.76 Mpps which equals the line rate of two 10 Gbit/s connections with minimum-sized packets. To simulate a realistic workload, one byte of each packet is modified. As a consequence, at least one byte is loaded into the L1 cache of the processor. We compare our implementation to the reference implementation in C (commit `d89d68b`), an implementation in C# (commit `484485b`) written by Maximilian Stadlmeier [18, 19] and an implementation in Go (commit `4145aa8`) written by Sebastian Voit [20]. All measurements have been performed on two single-ported Intel X510 (82599-based) NICs since they perform better than a dual-ported NIC (probably due to hardware limitations of the PCIe connection).

6.1 Throughput

We run the forwarding application at different CPU frequencies to measure the overall performance of our implementation and identify possible bottlenecks. Figure 6.1 compares the throughput of our implementation to ixy in C, C# and Go. The benchmark shows that the relative difference between the different implementations becomes smaller with increasing CPU frequency. The greatest performance difference between C and Rust is at 2.6 GHz where the C implementation is about one-eighth faster than Rust. At full CPU speed without overclocking, the difference between these two implementations drops to 3%. With dynamic overclocking, i.e. the Intel Turbo Boost feature that allows the CPU to accelerate to up to 3.6 GHz, there is no measurable performance difference between the C and the Rust version. To forward a packet, the C

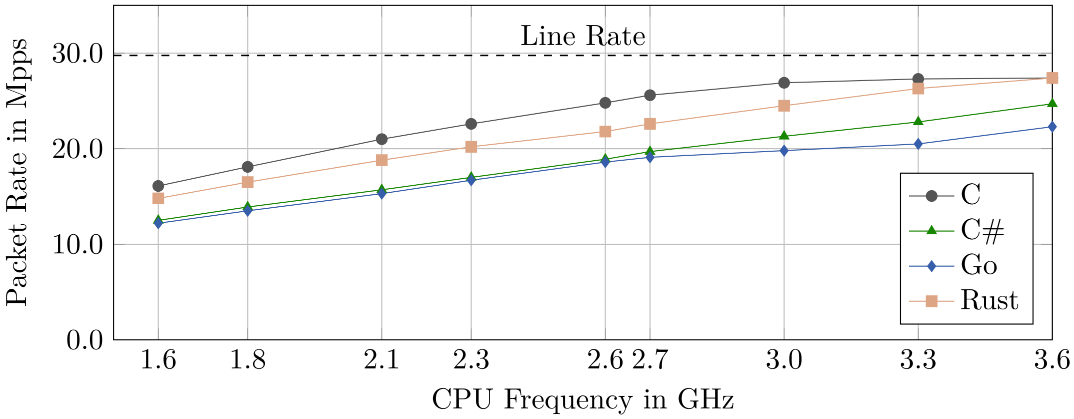


Figure 6.1: Bidirectional single-core forwarding performance with varying CPU speed and a batch size of 32 packets.

implementation requires a minimum of 100 CPU cycles, while the Rust implementation needs at least 108 CPU cycles.

Interestingly, not modifying the packets when forwarding them has a different impact on the implementations. While the C version is surprisingly slightly slower when not touching the packets at the native CPU clock rate of 3.3 GHz, the Rust version is at least 3% faster.

The main reasons for the high performance of our implementation are on the one hand the general driver design and on the other hand the used data structures. Avoiding memory allocations in the receive and send functions by using already allocated buffers in our memory pools and memorizing the physical addresses of all of these buffers instead of reopening `/proc/self/pagemap` every time a virtual address has to be translated to a physical address also make a huge performance difference.

Unlike the C version, we provide a safe interface regarding the packets by wrapping them in `Packet` structs. Unfortunately, moving structs instead of simple raw pointers has negative impacts on performance. We compensate for these performance losses by utilizing efficient data structures like `Vec<T>` and `VecDeque<T>` for all stacks and queues in our driver since they provide amortized costs of $\mathcal{O}(1)$ for the `push()` and `pop()`, respectively `push_back()` and `pop_front()` operations.

Before commit `6cbfac6`, the internal stacks and queues of the receive and transmit queues, i.e. `IxgbeRxQueue` and `IxgbeTxQueue`, used `Packet` structs to store all packets currently in use by the queues. Replacing the structs by a single `usize` referencing the corresponding buffer in the memory pool boosted performance dramatically by almost four million packets per second.

To prevent frequent resize and copy operations, all stacks and queues are allocated with the maximum required capacity where this size is known at allocation time. That is the case for all stacks and queues in our driver.

Due to the use of `Rc<RefCell<Mempool>>` for all references to our memory pools, accessing the pools is quite expensive. The reference counter has to be increased and decreased and the `RefCell<Mempool>` has to verify that there is only one mutable reference to a pool at a time (similar to a mutex). To reduce the number of accesses to the pools when cleaning the transmit queue, we enforce that packets sent through the same queue belong to the same memory pool. Hence it is sufficient to access the memory pool once for a batch of packets.

To improve the performance regarding the memory pool accesses, we tried to replace the reference counter by an ordinary reference (`&RefCell<Mempool>`) which required us to specify lifetimes for the references to the memory pool and all structs containing references or structs referencing the memory pool. Unfortunately, this prolonged the scopes of some values in an unexpected way due to yet-to-be-fixed bugs in the Rust compiler, namely issue 21906[1] regarding our `set_reg32()` method and issue 51132[2] regarding `reset_and_init()`. Hence, changing these references to normal references could be future work on the driver when these issues have been addressed.

6.2 Batching

Batching has a strong influence on the performance. Receiving and sending a packet requires a costly PCIe round-trip when accessing the queue index registers. Figure 6.2 and Figure 6.3 show how the performance is affected by different batch sizes at the lowest and the highest possible CPU frequency. Unfortunately, the C# implementation crashes with a batch size of 256 packets at 3.3 GHz, so we cannot tell if it could achieve the same packet rate as C and Rust.

It can be stated that batch sizes up to 64 packets per batch lead to a better overall performance for all implementations. Greater batch sizes than 128 packets per batch have a very little gain on performance as the number of cache misses increases, too. Hence, batch sizes should neither be chosen too small nor too large. For our implementation, a batch size of 32 or 64 packets per batch seems reasonable.

[1] https://github.com/rust-lang/rust/issues/21906

[2] https://github.com/rust-lang/rust/issues/51132

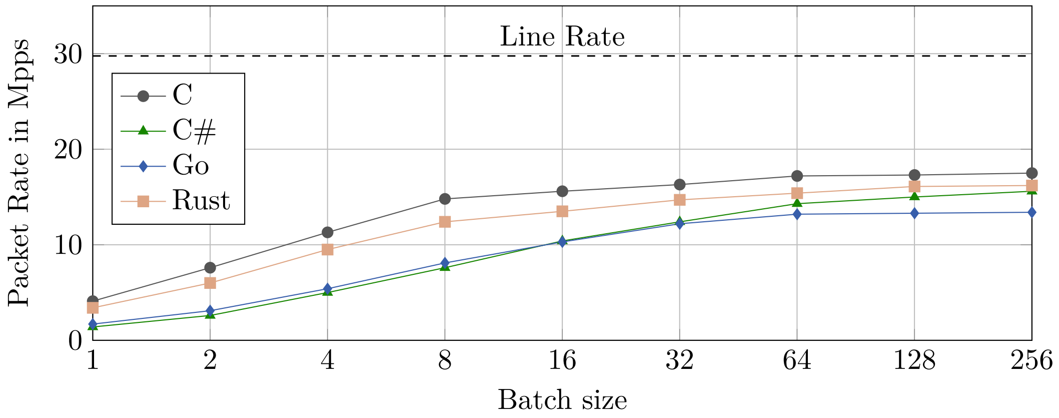


Figure 6.2: Bidirectional single-core forwarding performance with varying batch size at 1.6 GHz.

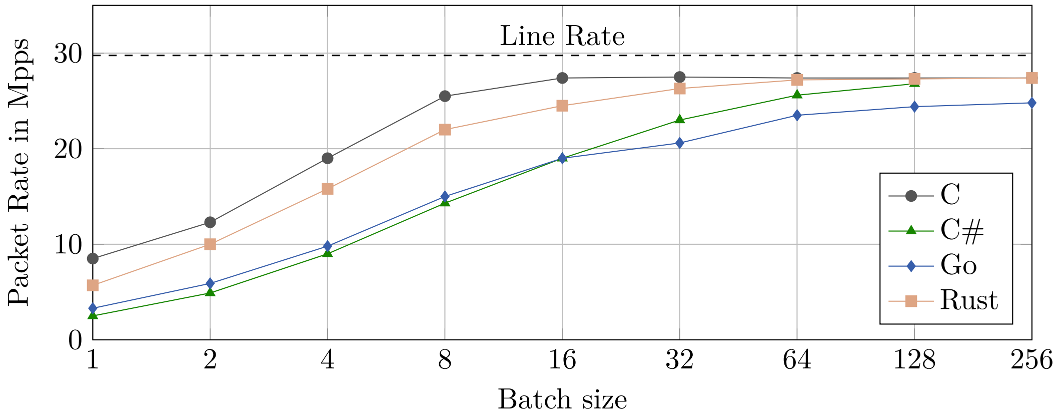


Figure 6.3: Bidirectional single-core forwarding performance with varying batch size at 3.3 GHz.

6.3 Profiling

Application	RX	TX	Forwarding	Memory Mmgt.
Rust forwarder	43.8	33.5	21.3	7.0
C forwarder	39.8	16.9	22.0	21.0

Table 6.1: Processing time in CPU cycles per packet.

To profile our forwarder application, we run `perf` at the minimum CPU speed of 1.6 GHz with the default batch size of 32 packets per batch to ensure that the CPU is the bottleneck. Table 6.1 shows how many CPU cycles are spent on which function and compares it to the C implementation.

In both drivers, the receive function is much slower as it has to fetch the data into L1 cache while all other functions operate on the cache. Overhead for memory management is significant in C, although it is still far lower than the 100 CPU cycles of the Linux kernel. Rust inlines many functions, so the seven CPU cycles for memory management have to be taken with a small grain of salt.

Looking at DPDK with 61 CPU cycles per packet, there is still room for improvements in our implementation. However, part of that truth is that DPDK supports almost all hardware offloading features at the price of increased complexity while ixy only uses CRC checksum offloading.

6.4 Unsafe Code

Our implementation of the ixy driver consists of a manageable number of lines of unsafe code. Table 6.2 compares the amount of unsafe code used in our implementation to the amount of unsafe code in two other Rust network drivers, the e1000 and the rtl8168 driver from Redox which are presented in more detail in Chapter 7.

Driver	Speed	Lines of Code	Unsafe Code	% Unsafe
ixy	10 Gbit/s	1306	125	9.57%
e1000 (Redox)	1 Gbit/s	393	140	35.62%
rtl8168 (Redox)	1 Gbit/s	362	144	39.78%

Table 6.2: Lines of unsafe code in three different network drivers written in Rust.

The results clearly show that our implementation uses proportionally less unsafe code than the two other drivers. In fact, while ixy consists of 10% unsafe code, e1000 and rtl8168 are made up of almost 40% unsafe code. Interestingly, the driver with the highest amount of code in total has the least amount of unsafe code while the driver with the least amount of code in total has the highest amount of unsafe code.

However, many lines of unsafe code are not necessarily an indicator for bad code quality or unsafe programs. Vast amounts of the Rust standard library consist of unsafe code and nevertheless Rust is considered a safe programming language. It is therefore necessary to assess on a case by case basis whether unsafe code is used appropriately or not.

In case of the Redox drivers, it stands out that they make massive use of `unsafe` code with about 150 lines of unsafe code out of about 400 lines in total. Looking at their source code, we suspect that the developers of Redox prioritized having working implementations of these standard drivers over minimal use of `unsafe` and providing safe abstractions, and thus declared too much code as `unsafe`.

We do however have to note that reducing the amount of unsafe code should be the goal of every developer as yet one erroneous line of unsafe code can be enough to cause undefined behavior.

Chapter 7

Background and Related Work

Rust is still a quite new programming language. Since the first stable release was published merely three years ago, there are not many Rust programmers and projects (especially in relation to low-level programming) yet. Nonetheless, at least three operating systems are currently developed in Rust: Blog OS [21], intermezzOS (which was heavily influenced by Blog OS) [7] and Redox [14]. Blog OS and intermezzOS are both educational operating systems that provide a lot of documentation to enable the reader to program an operating system himself. Redox on the other hand is a real-world operating system “aiming to bring the innovations of Rust to a modern microkernel and full set of applications” [14]. Redox also provides a lot of documentation but less detailed than the educational systems. However, as a working operating system Redox contains real-world drivers. Therefore, we will go into more detail on Redox in the following section.

7.1 Redox

Redox is an operating system written entirely in Rust. It was created by Jeremy Soller, first published on April 20, 2015 and is still actively developed by about 25 developers. Redox aims to be secure, free and usable by providing a fully functional Unix-like microkernel. It currently supports all x86-64 CPUs.

At this time, Redox contains drivers for NICs of the e1000 and rtl8168 families. It is interesting to read through the source code of these two drivers as they use different approaches and seem to be the only network drivers written in Rust so far. Unfortunately, there is not much documentation on the drivers yet. There is an online book about Redox written by the Redox developers, however, the chapter about drivers is still empty.

Looking at the source code, there are a few conceptual differences between the two drivers. While the e1000 driver uses constants for the register offsets of the NIC just like our implementation does, the rtl8168 driver uses a special register struct consisting of the registers defined as arrays of a user defined memory-mapped I/O (MMIO) type.

For reading from and writing to the registers, rtl8168 implements a `read` and a `write` method on the MMIO type while e1000 uses a `read_reg` and a `write_reg` function almost identical to our implementation.

Both drivers implement a device struct with methods like `new` to return a yet-to-be-initialized device, `init` to reset and initialize the device or `next_read` returning the size of the next packet in the receive queue. They also implement a trait `SchemeMut` for this device struct containing a `read` and a `write` function. Unlike our driver, these functions take a reference to a buffer and copy the packet data from DMA memory to that buffer and back. We can therefore say with certainty that these drivers are less performant than our implementation.

Nevertheless, these two real-world drivers prove that writing network drivers in Rust is not just an abstruse scientific idea but a practical and feasible task, one that has already been taken over.

Chapter 8

Conclusion

We implemented a user space network driver in Rust to show language specific advantages and disadvantages of Rust and to answer the question whether Rust is a suitable programming language for network drivers. The full code of our implementation as well as the code of the reference implementation are available on GitHub [3, 4].

As Chapter 3 makes clear, one of the most important requirements for a programming language for network drivers is performance. While this might not be the case for all kinds of drivers, for network drivers it certainly is. Thus, a reasonable driver should have a performance sufficiently close to an implementation in C or C++, two languages well known for their efficiency and widespread use. The performance measurements in Chapter 6 show that Rust is able to fulfill this requirement. Though our implementation is slightly less performant than the reference implementation in C, it is fast enough to be used in real-world applications as it is more than six times faster than the default kernel network stack.

The great advantage of Rust besides performance is its memory safety. Although, like any man-made object, Rust is not 100% perfect – there is a lot of unsafe code in the standard library and from time to time even bugs like a recent security vulnerability regarding `str::repeat()` shows[1] –, Rust does a great job on improving the safety of computer programs by enforcing strict rules on memory handling.

Regardless of the programming language, code contains on average 15-50 errors per 1000 lines of code [12]. There might not be less programming errors in Rust code but

[1] https://blog.rust-lang.org/2018/09/21/Security-advisory-for-std.html

at least they are less likely to lead to undefined behaviour and jeopardize our system like programs in C/C++ do.

Future work on the driver might include an implementation of the VirtIO-version as well as minor improvements regarding the overall performance. Additionally, a multi-threaded version of the ixgbe-driver could be implemented. Most of our code like the memory allocations on the huge pages should be thread-safe already so this might be an achievable goal for the near future.

Based on our findings we can conclude that Rust is a very well-suited programming language for writing network drivers. It is not only a fast and safe systems programming language but was also voted most beloved programming language in 2016, 2017 and 2018 [15, 16, 17]. Writing more drivers in Rust would certainly lead to safer and more reliable computer systems.

Chapter A

List of Acronyms

BAR	Base address register.
CPU	Central processing unit.
DMA	Direct memory access. Feature of computer systems that allows hardware to access main system memory independent of the CPU.
IOMMU	I/O memory management unit. Connects a DMA-capable I/O bus to the main memory.
MMIO	Memory-mapped I/O.
NIC	Network interface card.
OSI	Open Systems Interconnection. Reference model for layered network architectures by the OSI.

Bibliography

[1] Jim Blandy and Jason Orendorff. *Programming Rust: Fast, Safe Systems Development.* O'Reilly Media, 2017. ISBN: 1491927283.

[2] *DPDK Website.* `https://www.dpdk.org/`. Accessed: 2018-10-05.

[3] Simon Ellmann. *Ixy.rs source code.* `https://github.com/ixy-languages/ixy.rs`. Accessed: 2018-09-13. 2018.

[4] Paul Emmerich. *Ixy source code.* `https://github.com/emmericp/ixy`. Accessed: 2018-09-13. 2018.

[5] L Gorrie et al. *Snabb: Simple and fast packet networking.*

[6] *Intel 82599 10 GbE Controller Datasheet.* Rev 3.3. Intel. 2200 Mission College Blvd., Santa Clara, CA 95052, USA, Mar. 2016.

[7] *intermezzOS.* `http://intermezzos.github.io/`. Accessed: 2018-09-15.

[8] Taesoo Kim. *A minimal Linux kernel module written in rust.* `https://github.com/tsgates/rust.ko`. Accessed: 2018-10-03. 2016.

[9] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, 2018. ISBN: 9781593278281.

[10] Eddie Kohler et al. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[11] *Linux Kernel Documentation: VFIO - "Virtual Function I/O".* `https://www.kernel.org/doc/Documentation/vfio.txt`. Accessed: 2018-10-01.

[12] Steve McConnell. *Code complete.* Pearson Education, 2004.

[13] Ben Pfaff et al. "The Design and Implementation of Open vSwitch." In: *NSDI.* Vol. 15. 2015, pp. 117–130.

[14] *Redox.* `https://www.redox-os.org/`. Accessed: 2018-09-15.

[15] *Stack Overflow Developer Survey Results 2016.* `https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted`. Accessed: 2018-09-28.

[16] *Stack Overflow Developer Survey Results 2017.* https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted-languages. Accessed: 2018-09-28.

[17] *Stack Overflow Developer Survey Results 2018.* https://insights.stackoverflow.com/survey/2018#technology-most-loved-dreaded-and-wanted-languages. Accessed: 2018-09-28.

[18] Maximilian Stadlmeier. *Ixy.cs source code.* https://github.com/ixy-languages/ixy.cs. Accessed: 2018-09-21. 2018.

[19] Maximilian Stadlmeier. "Writing Network Drivers in C#". BA thesis. Technical University Munich, Aug. 2018.

[20] Sebastian Voit. *Ixy.go source code.* https://github.com/ixy-languages/ixy.go. Accessed: 2018-10-13. 2018.

[21] *Writing an OS in Rust (Second Edition).* https://os.phil-opp.com/. Accessed: 2018-09-15.