Safe Kernel Programming with Rust

JOHANNES LUNDBERG

Master in Computer Science
Date: July 21, 2018
Supervisor: Johan Montelius
Examiner: Mads Dam
Swedish title: Säker Programmering i Kärnan med Rust
School of Computer Science and Communication

Abstract

Writing bug free computer code is a challenging task in a low-level language like C. While C compilers are getting better and better at detecting possible bugs, they still have a long way to go. For application programming we have higher level languages that abstract away details in memory handling and concurrent programming. However, a lot of an operating system's source code is still written in C and the kernel is exclusively written in C. How can we make writing kernel code safer? What are the performance penalties we have to pay for writing safe code? In this thesis, we will answer these questions using the Rust programming language. A Rust Kernel Programming Interface is designed and implemented, and a network device driver is then ported to Rust. The Rust code is analyzed to determine the safeness and the two implementations are benchmarked for performance and compared to each other. It is shown that a kernel device driver can be written entirely in safe Rust code, but the interface layer require some unsafe code. Measurements show unexpected minor improvements to performance with Rust.

Sammanfattning

Att skriva buggfri kod i ett lågnivåspråk som C är väldigt svårt. C-kompilatorer blir bättre och bättre på att upptäcka buggar men är ännu långt ifrån att kunna garantera buggfri kod. För applikationsprogrammering finns det tillgängligt olika högnivåspråk som abstrakterar bort den manuella minneshanteringen och hjälper med trådsäker programmering. Dock fortfarande så är större delar av operativsystemet och dess kärna är endast skriven i C. Hur kan vi göra programmering i kärnan säkrare? Vad är prestandakonsekvenserna av att använda ett säkrare språk? I denna uppsats ska vi försöka svara på dessa frågor genom att använda språket Rust. Ett programmeringsgränssnitt i Rust är implementerat i kärnan och en nätverksdrivrutin är portad till Rust. Källkoden skriven i Rust är analyserad för att bedömma säkerheten samt prestandan är jämförd mellan C och Rust implementationerna. Det är bevisat att vi kan skriva en drivrutin i enbart säker Rust om vi kan lita på några osäkra funktioner i gränssnittet. Mätningar visar lite bättre prestanda i Rust.

Contents

Chapter 1

Introduction

1.1 Operating system's kernel

In an operating system, everything is built around the kernel and the kernel manages all interaction between the user, the software applications and the hardware. Therefore it is important that the kernel is highly optimized or the whole system will suffer from bad performance. To accomplish this, a powerful, low-level language is required, and for this reason, the kernel is usually written in C. In C, the programmer have full control and are expected to manage everything when it comes to memory safety. This has some problematic consequences described in the next section.

1.2 Problem description

In userland, applications run in isolated compartments and one ill-behaving application will seldom affect other applications. However, even if the applications can be written in a safer, higher level language it is a false sense of safety because they all depend on the kernel. The monolithic kernel is essentially one big C program without isolated parts where one small bug in one place can cause the whole system to fail[1]. Some of the common bugs in C code are:

- **Double free** - Freeing memory that has already been freed.

[1] The micro kernel design tries to solve this by isolating different parts of the kernel from each other but it doesn't solve the underlying problem.

- **Use after free** - Access memory that has been freed.
- **Data races** - Failure to use proper locking mechanism.

In the best case, any of these bugs in the kernel would cause the system to crash. In the worst case, they would cause subtle errors that are difficult to detect and that would cause all kinds of weird behavior in the operating system. One could almost say, the whole operating system depends on a very fragile core, similar to a house of cards.

1.3 Purpose

Kernel programming is inherently unsafe and making it safer is not an easy task since the code is running close to the hardware and has to be highly optimized for performance. In the past, there have been a few attempts to make kernel programming safer. This includes things from language customization with trusted compilers and type safe assembly language to transactional semantics for system calls.

In recent years a new possible way of making kernel programming safe has surfaced. This is the programming language Rust. Among other features, Rust has built in static analysis based on ownership that catches bugs that would otherwise go unnoticed in a language like C. It is a modern alternative to C and C++ that claims to prevent many of the bugs common to C like languages. The purpose of this thesis is to evaluate Rust for safe kernel programming. A single device driver will be ported to Rust to test the thesis. Further porting of other device drivers or kernel interfaces is left as possible future work.

1.4 Goals

The goal is to determine if it is feasible to use Rust as a safer alternative to C in kernel level programming. To answer this question we have to look at two different aspects.

Safeness

Rust comes in two flavors, safe Rust with its strict compiler rules and static analysis and unsafe Rust, which unlocks the power of raw pointers and the risk to "shoot oneself in the foot". To be able to interface with foreign languages and do low-level manipulation of memory, Rust has the ability to compile code in an unsafe mode where it permits the use of raw pointers and other things regarded as "unsafe". Unsafe code is required when writing low-level code, typically found in the kernel, but it should be kept to a minimum. For reference, Rust itself uses unsafe code internally to wrap things like raw pointers in safe interfaces. This thesis should show how much unsafe code is required to write a functional device driver in Rust.

Performance

As previously mentioned, performance is critical in the operating system's kernel and memory safety comes at a price. Since the Rust compiler will add things like boundary checks to its smart pointers, there will be some performance penalty. This thesis should show how much the performance differ between a native C implementation and an equivalent Rust implementation.

1.5 Methodology

The methodology can be divided into the following steps, listed in chronological order.

Cross-compile Rust

Rust by default comes with support for cross compiling to different target operating systems and architectures. It does not however support a kernel target out of the box so some manual setup and configuration is required. Among other, a LLVM target configuration for the kernel has to be written. A RustKPI (Rust Kernel Programming Interface) is created as a loadable kernel module by reusing a lot of the code found in Rust's standard library.

Porting

Since the kernel is written in C there exists no device drivers written in Rust. To be able to compare performance between Rust and C, one device driver is ported[2] from C to Rust. The kernel consists of various subsystems and device drivers, many that could be a potential porting target for this thesis. Some requirements are:

- **Performance** - It should be something that can generate high CPU load and be benchmarked.
- **Few dependencies** - The work load should be reasonable, i.e. too many dependencies to different kernel programming interfaces would be too time-consuming to port.
- **Comparison** - The Rust implementation should be comparable to the original C implementation to give fair performance comparison results.

After considering systems like USB and storage I/O, it was decided that a suitable candidate for porting is Intel's Ethernet network card driver, "e1000". This driver depends only on a few kernel programming interfaces apart from IfLib and no other driver or system depends on it. As a network card driver it can also be pushed to generate high CPU loads. The driver interfaces with RustKPI and is cross-compiled for the kernel the same way.

Analysis

Whenever unsafe Rust is used in the code it has to be explicitly marked with an `unsafe {...}` block. The device driver code will be analyzed for any usage of unsafe and the code within each unsafe block further divided into different categories, each category being one action that Rust regards as unsafe. The safe code will be assumed to be safe as guaranteed by the Rust compiler.

Benchmarks

Iperf [1] is used to benchmark the performance. Iperf is a tool for network performance measurement and it has both client and server functionality. Iperf runs a duration of ten seconds and outputs the average bitrate and number of bytes transferred. To analyze deeper the CPU utilization of Rust and C implementation, CPU utilization of

[2] Porting is when source code is re-written from one language to another, or from one operating system to another.

kernel threads are measured during maximum transfer rate between a Bhyve virtual machine and its host.

1.6 Limitations

In this thesis, the focus is on the device driver code and a few assumptions are made regarding the surrounding environments.

Unsafe code

Some calls to C functions and Rust unsafe functions will be wrapped in a safe interface for convenience. It is assumed that the input can be trusted so that these are considered as safe and not counted as unsafe code for each invocation.

Rust compiler

It is assumed that the Rust compiler is bug free and can keep its guarantees regarding memory safety.

Kernel managed objects

The kernel is managing some objects used by the Rust code. It is assumed that all pointers to these objects handed to the Rust code by the kernel are correct and valid.

Performance of Rust code

Since e1000 was ported to IfLib[3], a lot of the heavy lifting is now done by IfLib and the driver code is mostly in charge of keeping an internal state and manipulating hardware registers. This will likely reduce the possible performance difference in C vs Rust implementations. Porting IfLib to Rust as well would probably give a more interesting performance comparison.

[3] As of FreeBSD 12 code common to network device drivers have been moved to a shared library, IfLib, to simplify device driver development.

Chapter 2

Background

2.1 Operating system

Most computers except for the smallest embedded devices run some kind of operating system. Popular operating systems on desktop and laptop computers are Windows, macOS and Linux. Also available on personal computers but more popular on servers are the BSD Unix derivatives FreeBSD, OpenBSD, NetBSD and others. Windows and macOS are proprietary and thus not suitable for this purpose. Linux and the BSD derivatives are open source, which means that all of the software that comes with the operating system is freely available. Although any open source operating system could be used, in this thesis the most common of the BSD derivatives, FreeBSD [2] is used.

2.2 Kernel and userland

A common way to design an operating system is by using a monolithic kernel. In a monolithic kernel, the kernel is essentially one big program, usually written in C. External modules can be loaded during runtime to add support for hardware devices or add new services.

Userland is the part of the operating system that exists outside of the kernel. Here is where the users' applications are executed in an unprivileged mode. Access to the kernel and hardware is privileged and is done through system calls to the kernel, which runs in privileged mode. In modern processors privileged mode is controlled by hardware to increase security.

Normally applications execute in userland where they are kept isolated from each other and the kernel. There are various security measures in place to protect a system against ill-behaving applications.

2.3 Consequences of software bugs

Writing bug free computer code is a challenging task in a low-level language like C. While C compilers are getting better and better at detecting possible bugs, they still have a long way to go. For application programming we have higher level languages that abstract away details regarding memory handling and concurrent programming. However, a lot of an operating system's source code is still written in C and the kernel is exclusively written in C.

It's worth mentioning the gravity of the problems that software bugs can cause. For example:

The Northeast blackout of 2003

A race condition in the software caused a several days long electric blackout in Northeast USA. [3]

Therac 25

A race condition in the software of a radiation therapy machine caused the machine to give massive overdoses of radiation. The bug caused several deaths. [4]

2.4 Bugs in the kernel

While there are many easily accessible solutions for userland applications, like using higher level languages, to help avoid these kinds of accidents, that is not the case for the kernel. Some existing techniques used to prevent bugs in the kernel are static analysis by the compiler, code review and manual testing. None of them can make any guarantees about memory safety and kernel developers are in need of better tools to write safe code.

There has been attempts to solve this in the past and some proposals are presented in the related work section. In this thesis the Rust programming language [5] is used as a way to enforce safe programming in the kernel and the goal of this thesis is to determine if Rust is feasible to use for safe kernel programming.

2.5 Rust

Rust was first developed by Graydon Hoare in 2010 but since shortly after its introduction it is developed by Mozilla's Rust developer team. Rust was designed to eliminate many of the bugs common to C and C++ while not sacrificing performance.

As stated on Rust's website [5]

> Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.
>
> Features
>
> - zero-cost abstractions
> - move semantics
> - guaranteed memory safety
> - threads without data races
> - trait-based generics
> - pattern matching
> - type inference
> - minimal runtime
> - efficient C bindings

Rust is a language that is capable of both high level programming of applications as well as low-level systems programming. It does not depend on garbage collection so it is also possible to use in the kernel, as proven by Levy et al., who built a kernel in Rust [6] and the Redox team who are writing an operating system completely in Rust, Redox [7]. The Rust book [8] is a comprehensive resource for anyone looking for more information about Rust and for more deeper background information there also exists many research papers that has influenced the design of Rust. For example, Rust's type system is influenced by the Cyclone project [9] where the C programming language is made safer by using statically-scoped regions and tracked pointers. Tracked pointers include unique pointers, something that is central to Rust. They also found a great synergy between regions and tracked pointers and they use the LIFO region machinery to support borrow pointers, also something central to Rust. In the paper External Uniqueness is Unique Enough [10], Clark and Wrigstad show with ownership types that an external reference is unique while at the same time permitting internal references.

For parallel processing and safe concurrency Rust uses what is called Three Layer Cake model [11]. In this model message passing is used for I/O in the top layer. The middle layer use fork/join for divide and conquer. Finally the bottom layer use SIMD to parallelize computations even further where possible. In Rust's bibliography also listed are, among other, Microsoft Research's Singularity project papers [12] and [13]. These papers describe how safe concurrency is achieved in Singularity OS through isolated processes and message passing with channels, something that is a core feature in Rust.

Rust safe code

The default mode for Rust is "safe Rust". Safe in this context meaning "guaranteed memory safety" and "free of certain class of bugs", not safe as in sandboxed or with limited API access. The compiler makes assertions at compile time to ensure memory safety. Rust can ensure memory safety without using garbage collection because of its central feature, ownership. Here follows a short description of some of Rust's features and terminology that is important to grasp in order to understand how the language works.

Ownership

In Rust, every object can have exactly one owner[1]. An object can not be moved out of an ownership but can be copied if it implements the Copy trait.

Move semantics

If a variable, of a type that does not implement the Copy trait, is assigned to another variable the ownership is transferred and the old

[1] A variable owns the data or object it holds or points to. Objects can be shared by using higher level constructs like smart pointers.

variable becomes invalid. Any attempt to access the old variable will result in a compile error. Primitive types implement the Copy trait.

Borrow

To share an object one has to borrow a reference to it. Multiple immutable references can exist simultaneously. A borrow will end when the reference is dropped, i.e. goes out of scope.

Mutable borrow

Declaration of a new variable or borrow of a reference is always immutable unless explicitly stated to be mutable. For mutable references, there can exist only one at any given time to prevent data races or inconsistent states. Any attempt to borrow another mutable reference while one already exists will result in compile error.

Lifetimes

References are bound by lifetimes. In most cases, these are inferred by the compiler but sometimes one has to explicitly assign a lifetime to a borrow to tell the compiler how long that reference will exist so the compiler can make the correct analysis.

Rust unsafe code

The Rust book refers to unsafe Rust as a second language. There are two reasons why unsafe Rust exists. One is that static analysis is overly conservative. Sometimes the Rust compiler will play it too safe and the programmer needs to bypass its strict rules by using unsafe code. The other is because the underlying hardware in computers is inherently unsafe and Rust needs to allow the programmer to access hardware or do near hardware programming. These actions are unsafe in Rust:

- Dereference a raw pointer.
- Call an unsafe function or method.
- Access or modify fields in a union.
- Access or modify a mutable static variable.

- Implement an unsafe trait.

Raw pointers are frequently used when interfacing with C code or implementing low-level building blocks. It is important to keep in mind the dangers with raw pointers compared to references or smart pointers. Raw pointers

- are allowed to ignore the borrowing rules and have both immutable and mutable pointers, or multiple mutable pointers to the same location.
- aren't guaranteed to point to valid memory.
- are allowed to be null.
- don't implement any automatic clean-up.

It should be mentioned that even within an unsafe block, the compiler still checks references for memory safety. Unsafe code does not disable any compiler functionality, it enables extra functionality. For details on how unsafe code is used in practice, see the next section.

C vs Rust comparison

Let's look at two practical examples on how Rust eliminates some common bugs.

Dangling pointer

```
int *return_reference() {
    int i = 0;
    return &i;
}
```

Listing 2.1: Dangling pointer in C

```
fn return_reference<'a>() -> &'a i32 {
    let i: i32 = 0;
    &i
}
```

Listing 2.2: Dangling pointer in Rust

This is just one of many examples of situations where one can end up with dangling pointers. While a modern C compiler will give a warning for returning a reference to a local variable, it will happily ignore it and compile the code in listing 2.1 to an executable that will crash on execution. The Rust code in 2.2 will fail compilation with the following error:

```
error[E0597]: `i` does not live long enough
 --> src/main.rs:4:6
  |
4 |     &i
  |      ^ borrowed value does not live long enough
5 | }
  | - borrowed value only lives until here
```

Listing 2.3: Dangling pointer in Rust. Compile error

The Rust borrow-checker does not allow references to data where the reference will outlive the data.

Multiple pointers to same data

This can lead to several bugs like use after free and data races.

```
void main() {
    int *x = malloc(sizeof(int));
    int *y = x;
    printf("%d␣%d\n", *x, *y);
}
```

Listing 2.4: Multiple pointers in C

```
fn main() {
    let x: Box<i32> = Box::new(0i32);
    let y = x;
    println!("{:?}␣{:?}", x, y);
}
```

Listing 2.5: Multiple pointers in Rust

Here the C compiler allow free cloning of pointers. Listing 2.4 is not an error by itself but it shows that it is easy to create multiple *mutable* pointers to the same data which the programmer has to keep track of. This situation can easily lead to bugs like use after free if `free()` is called on one pointer while the other is still in use. In Rust, the ownership model does not allowed this to happen and the code in listing 2.5 will cause the following compile error:

```
error[E0382]: use of moved value: `x`
 --> src/main.rs:4:27
  |
3 |     let y = x;
  |         - value moved here
4 |     println!("{:?} {:?}", x, y);
  |                           ^ value used here after move
  |
  = note: move occurs because `x` has type `std::boxed::Box<
     i32>`, which does not implement the `Copy` trait
```

Listing 2.6: Multiple pointers in Rust. Compile error

Since `Box<i32>` does not implement the Copy trait, the ownership of the data is moved to the new variable and the old variable becomes invalid.

2.6 Related work

In this and related research papers the word “safe” is used in situations like “safe code” or “safe programming” and this can have different meanings. As we will see in this section, some research refer to ”trusted code” and some refer to it as “well-behaving” or “bug free” code. In a few papers they also overlap.

Related work using Rust

Redox

Redox [7] is an operating system entirely written in Rust and it’s using micro kernel design for its kernel. Redox doubles up on safety by combining a micro kernel where device drivers run outside of the kernel and the benefits of a safe language like Rust.

System Programming in Rust: Beyond Safety

In [14] Balasubramanian et al., explore the benefits of Rust beyond safety. They argue that Rust enables implementation of capabilities not possible in other languages. Specifically, zero-copy software fault isolation, efficient static information flow analysis and automatic checkpointing. These capabilities is actively researched but the cost of implementing them has been high. The conclusion they arrive to is that

Rust will enable these capabilities to be commoditized. These capabilities, especially software fault isolation, is also relevant in the kernel. For example in device drivers like described in [15] by Herder et al., but with zero-copy.

The Case for Writing a Kernel in Rust

In the paper Ownership is Theft: Experiences Building an Embedded OS in Rust [16] suggested some improvements to Rust that would enable writing an operating system for embedded devices. Later, in [6] Levy et al. mention that while previous efforts in writing a kernel in Rust has required changes to Rust, this paper reaches another conclusion. That is, unmodified Rust is enough and in addition, very little unsafe code is needed. They describe how the kernel mechanisms DMA, USB and buffer caches can be implemented using Rust.

Safe, Correct, and Fast Low-Level Networking

In low-level networking stacks performance is commonly prioritized over safety. In [17], Clipsham attempts to change this by implementing a network stack in Rust that runs in userland, using the netmap API [18]. Clipsham designed a domain specific language so the code needed to parse packets was significantly reduced. By building a safe interface to unsafe APIs, Clipsham show that it is possible to provide a high performance network stack without sacrificing safety.

Other related work

Safe Kernel Programming in the OKE

OKE, Open Kernel Environment, allow non-root users to load native code in the Linux kernel. The implementation consists of three parts, the code loader, the custom compiler and the legislator. The Linux kernel module loader tools (insmod and rmmod) are extended with functionality that accepts a binary blob, together with authentication and credentials. The compiler takes as input the kernel module source code, user's credentials and customization rules, and outputs the compiled kernel module, an identify file and a MD5 checksum. The legislator is an external tool that generates the rules used by the compiler. [19]

TALx86: A Realistic Typed Assembly Language

Typed Assembly Language (TAL) is a method of making low-level programming safer. TALx86 [20] is a version for the Intel x86 processors. In TAL each value used by the code is annotated with a type. This information can be used to for example enforce type safety. A C-like language called Popcorn it also developed. Popcorn compiles to TALx86 and can be thought of as a safe dialect of C.

In his Master's thesis [21], Maeda use TAL and Popcorn to reduce system calls by moving the code from userland into the kernel and run it safely in kernel space.

Efficient and Safe Execution of User-Level Code in the Kernel

This project has two goals. The first goal is to improve performance by reducing system calls. This is done by running parts of the code in kernel mode or creating new, more efficient system calls. The second goal is to ensure that user-level code that runs in the kernel is done so in a secure way. For this various software- and hardware-based techniques like runtime monitoring of memory buffers, reference counters and spin-locks are used. Performance improvements as high as 80% could be seen and kernel safety checks show an overhead of only 2%. [22]

P-Bus: Programming Interface Layer for Safe OS Kernel Extensions

P-Bus introduces a new programming interface on top of the Linux kernel. New extensions are verified with a model checker to ensure the safety of code run in the kernel. A network driver was implemented using P-Bus. [23]

xCalls: Safe I/O in Memory Transactions

The xCall interface is a new API that provides transactional semantics for system calls. It's implemented for the Intel Software Transactional Memory compiler. Despite the overhead of implementing transactions in software, tests showed that transactions with xCalls improved the performance of two applications with poor locking behavior by 16% and 70%. [24]

Chapter 3

Methodology

The FreeBSD kernel is written in C and to interface with the kernel using Rust a compatibility layer is required. For this project this is called RustKPI - Rust Kernel Programming Interface. The device driver then uses RustKPI to interact with the kernel and the hardware. This is achieved by designing and implementing two kernel loadable modules, one for RustKPI and one for the device driver.

Figure 3.1 shows a block diagram of the RustKPI design layout. IfLib is lifted out from the kernel to emphasize the driver's dependency on it as well as the fact that it is a new addition to the FreeBSD kernel as of version 12.

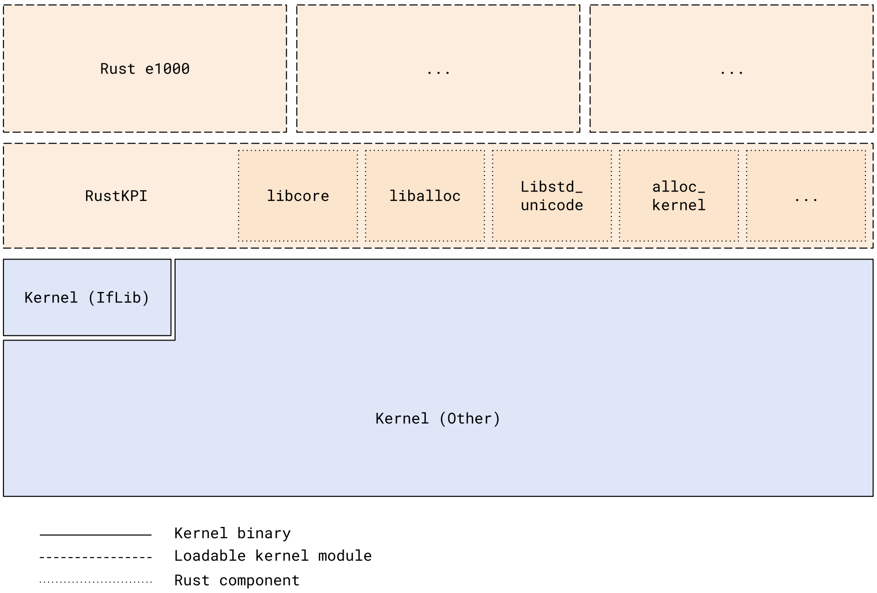

Figure 3.1: Kernel block diagram.

3.1 Cross-compile Rust

Normally Rust applications and libraries are compiled using a tool called Cargo. Cargo fetches and compile all dependencies automatically. Xargo[1] extends the functionality of Cargo by also managing sysroots for cross compiling. However, for this project a higher degree of control is desired so cross-compiling is done with the Rust compiler, "rustc", and Unix makefiles directly to allow full control.

To be able to compile Rust for the kernel the Rust compiler is configured to emit not only the standard Rust library files but also object files. The Rust library files are needed to satisfy dependencies for the compiler during compilation but only the object files are linked together to a kernel module. The linking is not different from how it's done in C, using a standard kernel module Makefile.

Userland applications are usually built with Rust's standard library, the std crate[2]. Since std depends on OS-specific functions it can not be used in the kernel. The std crate consists of many different sub-crates but only a few are needed. These are modified if needed and cross-compiled into a kernel crate. More details on this in the next chapter.

3.2 Port driver to Rust

To be able to benchmark the Rust code and compare performance to the native C implementation a device driver is ported to Rust. This should preferably be a driver that is a leaf node in the dependency graph, i.e. does not have any other device depending on it. It should also be a device that can generate enough CPU load that it will reach the bounds of the CPU's performance. For this project, the e1000 driver was chosen for porting to Rust. e1000 is a device driver for Intel's Ethernet network cards up to 1 Gb transfer rate. This driver supports a wide range of devices commonly found in laptops and desktop computers. For this purpose, a support for a subset of devices is ported to Rust.

First, all code needed to support the 82545EM device is ported. This is the hardware emulated by the hypervisor bhyve [25] and to

[1] Xargo has been a great help to get started but its future is uncertain as of 2018. Its functionality might be merged into Cargo.

[2] In Rust, a program or library source code package is called "crate"

target that first will make initial porting and testing easier. Development was done on a laptop running FreeBSD 12 and testing was done in a bhyve virtual machine on the same laptop.

Later, support for I218-LM and I129-LM devices is added. This is the hardware found in the laptops later used for hardware performance evaluation.

3.3 Analysis

Safeness

The Rust code is analyzed and blocks and lines of unsafe code are counted. Inside an unsafe code block, there are no guarantees from the compiler that things like raw pointers are valid so the amount of unsafe code should be as low as possible. One drawback with this method is that we have to trust the Rust compiler and assume the code is safe and bug free. Also, even if the developer knows that the code is safe, the Rust compiler is conservative and might not agree and still require an unsafe block. At any rate, it should give a good indication on safeness of the code.

Unsafe code used in the driver code can be divided into the following categories.

- **Function Call** - Call to unsafe C or Rust function. All calls to a Foreign Function Interface (FFI) are unsafe since the Rust compiler can not make any guarantees for what is happening beyond what it can see. Some functions in the Rust std or core libraries are unsafe as well, usually functions that involve manipulating raw pointers.

- **Unions** - Unions are structures where fields share the same storage. Unions are practical when sharing structures and e1000 support many devices that share the same code. Rust support unions but they are unsafe to access because the Rust compiler can not statically know which field is the active field. Because the unions used are internal to the e1000 driver code, they can easily be replaced with a safe Rust alternative like Enumerations. However, for simplicity in this project the C unions are converted to Rust unions.

- **Raw Pointers** - Dereferencing raw pointers.

Performance

Iperf [1] is used to transmit TCP/IP packets at maximum rate while CPU usage is monitored. This is done for both the original driver written in C and the Rust version. Performance of the two drivers is compared to see if using Rust is causing any performance penalties. One problem when doing this on the real hardware is that the traffic will be limited by the hardware or the TCP/IP connection between devices. In order to examine the device driver's properties at 100% CPU utilization traffic between a Bhyve virtual machine and its host is also benchmarked.

To produce flamegraphs[3], the device drivers were profiled with Dtrace and HWPMC (Hardware Performance Monitoring Counter), however, these utilities were not able to trace any Rust functions. Some attempts were made to configure the compiler and linker to enable profiling of the Rust implementation, but all were unsuccessful.

[3] A flamegraph can show graphically how much time the CPU spend in each function.

Chapter 4

Porting a network device driver

4.1 Setup

Hardware

Listed in B.2 are details regarding the two physical computers and the virtual machine used in development, testing and benchmarking. These are the machines involved in development, testing and benchmarking.

Computer A is used for benchmarking Iperf on real hardware and as Bhyve host for Computer B.

Computer B is a bhyve virtual machine hosted under Computer A. By default a bhyve virtual machine uses virtio-net as Ethernet device but it also has the option to emulate the Intel 82545EM hardware supported by the e1000 driver which is used in this thesis.

Computer C is used as support server and client when benchmarking Iperf on real hardware on Computer A but it's not part of the results.

Software

FreeBSD can be freely downloaded from FreeBSD's website [26]. It includes both binaries and source code for the complete operating system. This thesis requires at least FreeBSD 12-CURRENT as of May

2018 or later due to IfLib dependency. Any additional programs like Iperf can be install using ports [27].

Bhyve is included in a standard FreeBSD installation. It has many options so there exists a helper script in FreeBSD at location
`/usr/share/examples/bhyve/vmrun.sh`
which was used, with the exception that e1000 and not virtio-net was set as default network driver.

Rustup is the default installer tool for Rust. It is used for installing and managing multiple toolchains and switching between them on a per-project basis. For this thesis, Rust 1.25 nightly toolchain and source code are used. The reason for this is simply that it was the latest version when this work was started.

Bindgen is a tool that generates Rust bindings from given C header files. It significantly reduces manual labor when interfacing with large C projects in Rust. Bindgen can be installed with `cargo` that comes with the Rust installation.

4.2 RustKPI

Rust's compiler is based on the LLVM compiler infrastructure [28]. It includes many different target configurations for various operating systems and architectures but not for an uncommon target like the kernel. Therefor one must provide a custom configuration file in JSON format that enables and disables the correct features, like floats, mmx, etc, for the kernel target. The one used in this thesis is listed in B.3.

Rust crates

Rust's source code contains many sub-crates but only a few of them are needed for basic functionality in RustKPI:

- **core** - Provides the core functionality of Rust
- **std_unicode** - Dependency of alloc. Unclear if this is really needed in the kernel.

- **alloc** - Provides typical heap-related things like Box, Vec, etc. Requires an implementation of the allocator API.

Additional custom crates outside of Rust std:

- **alloc_kernel** - An allocator backend for the alloc crate that uses kernel malloc functions. Developed specifically for this thesis.
- **spin** - Provides Mutex and RWLock backed a simple spin-lock. Available on `crates.io`[1].
- **interpolate_idents** - This compiler plugin enables concatenated identifiers in macros, something Rust is lacking but is common in C. Available on `crates.io`.

A crate named **kernel** is created to wrap and re-export all the functionality contained in these crates, similar as is done in the **std** crate. To package this as a loadable kernel module, yet another crate called **kmod** is created. The kmod crate contains only a callback function for module load and unload events and some macro invocations that setup the static module data so that the kernel module can be loaded into the kernel and registered.

4.3 Rust e1000

The e1000 driver support many different devices and only support for a subset is needed for this thesis. This significantly reduced the amount of code needed porting. A complete listing of files in the C implementation can be found in B.4. The files that were completely or almost completely ported to Rust are:

- **if_em.c** - The entry point of the driver.
- **e1000_api.c** - Wrappers for device specific functions.
- **e1000_82540.c** - Most of the code for 82545EM is here.
- **e1000_ich8lan.c** - Most of the code for I218-LM and I219-LM is here.
- **e1000_nvm.c** - Hardware access functions.

[1] `https://crates.io` is a crate repository for Rust

- **e1000_mac.c** - Hardware access functions.
- **e1000_mbx.c** - Hardware access functions.
- **e1000_phy.c** - Hardware access functions.
- **e1000_osdep.c** - OS specific functions.
- **em_txrx.c** - Transmit and receive functions.

In addition, a few functions from other files are also ported and Rust bindings are generated from the C header files using bindgen.

Kernel modules rely heavily on C pre-processor macros to declare static structures used by the module loading mechanism. Expanding these macros reveal a mess of static C structures, with mutable pointers making circular dependencies, something that Rust does not allow. For the RustKPI kernel module it was fairly straightforward and the module could be completely implemented in Rust using Rust macros converted from C macros. For the e1000 driver code it was more difficult due to use of several mutable static structures. It could probably be solved with more time but for this thesis a bridge file was created in C where static structures are declared with C pre-processor macros, compiled with a C compiler and linked with the remaining Rust code to make the final kernel module. In addition, wrappers are created for some functions like PCI access functions that are in fact C pre-processor macros.

Starting with the device register callback which is the device driver entry point, the C functions were ported one by one to Rust for the 82545EM driver. After the 82545EM driver was tested and functional, the I218-LM and I219-LM drivers were ported.

Chapter 5

Evaluation

For evaluation, two things are looked at. How much unsafe code is required and what is the performance compared to the C implementation. If too much[1] unsafe code is required it would defeat the purpose of using safe Rust and one might as well stick to C. Performance is critical in the kernel since it serves the entire operating system and the hardware. A small performance penalty is acceptable considering what is gained by using a safe programming language.

5.1 Unsafe Code

Breakdown of device driver Rust code. This is not the final version of this code but it's at a working state, fully implemented. There is still room for improvement that would further eliminate usages of unsafe code.

- 9897 lines of code (excluding comments, blank lines and bindgen generated bindings).
- 430 lines of unsafe code that contain:
  - 56 calls to C functions.
  - 43 calls to unsafe Rust functions.
  - 73 access to unions.
  - 37 dereferencing raw pointers.

[1] One has to look at the system as a whole and determine whether it is acceptable or not.

Apart from only a few unsafe blocks where several accesses to union fields are lumped together in one unsafe block, all unsafe blocks are only one (Rust) command or one function call. In the following sections, a closer look is taken at each of the unsafe code categories.

Unsafe C functions

```
impl PciDevice {
    pub fn pci_read_config(&self, reg: u32, width: u32) ->
        u32 {
         let child: *mut device = self.inner.as_ptr();
         let parent: *mut device = unsafe { device_get_parent(
             child) };
         let ret = unsafe { rust_pci_read_config(parent, child
             , reg as c_int, width as c_int) };
         ret
    }
}
```

Listing 5.1: Example of calling unsafe C functions.

In this example two unsafe functions are called, `device_get_parent()` and `rust_pci_read_config()`. `device_get_parent()` is a kernel function to get the parent PCI device. `rust_pci_read_config()` is a wrapper for the C pre-processor macro `pci_read_config()`. Unsafe uses should generally be wrapped in safe interfaces if possible. Here `self.inner` is a raw pointer wrapped in a `NotNull` structure so it is confirmed to be not null and the kernel is trusted to provide a valid device pointer. To make the safe interface safer, some assertions or boundary checks could be added to the function arguments as well as checking if the parent pointer is null.

As we've seen, many of the unsafe calls to C functions can be wrapped in a safe interface. As a metric, this is quite volatile. If for example PCI access functions were not wrapped in safe Rust functions, this value would be higher. Further abstraction of unsafe C functions would reduce this value. For example, all access to C functions could be wrapped in safe interfaces with proper assertions to make them truly safe. This would reduce unsafe calls to C functions to zero.

Unsafe Rust functions

```
let vaddrs: *mut caddr_t = <raw pointer to storage>;
let count: usize = <number of elements>;
let vaddrs_slice: &[caddr_t] = unsafe {
    kernel::slice::from_raw_parts(vaddrs, count)
};
```

Listing 5.2: Example of calling unsafe Rust functions.

`slice::from_raw_parts()` is a function that takes a raw pointer and a length and with that creates a Rust slice. A slice is a “window” into a region of memory, usually an array of objects. It has built-in boundary checks and iterator functionality.

Similarly as C functions, some uses of unsafe Rust functions have been wrapped in safe functions where it is trusted that the function will be safe in that specific use. As long as the programmer can guarantee that the internal implementation is safe, the use of unsafe Rust functions can be hidden inside a safe interface. In this example, this code could be wrapped in a safe interface where the pointer is checked to be valid and count of elements is within a known valid range. Unsafe blocks can not be eliminated from the interface layer but from the device driver perspective, the use of unsafe Rust functions could probably be eliminated.

Unions

```
pub union DevSpec {
    /* Device specific structure used by 82545EM */
    pub _82541: DevSpec_82541,
    /*
     * Device specific structure used by I218-LM
     * and I129-LM
     */
    pub ich8lan: DevSpec_ich8lan,
}
unsafe {
    let srs: &mut [ShadowRam; 2048] = &mut adapter.hw.
       dev_spec.ich8lan.shadow_ram;
    for sr in &mut srs.iter_mut() {
        sr.modified = false;
        sr.value = 0xFFFF;
    }
}
```

Listing 5.3: Example of unsafe access to Union.

For simplicity and because of time constraints, all C unions were converted to Rust unions. This does increase the use of unsafe code, however, all unions are internal to the driver code and can be replaced with a safe Rust alternative like Enumerations. This would eliminate all need for unsafe access to unions.

Raw pointers

```
let iflib_ptr: *mut iflib_ctx = <raw pointer from iflib>;
let adapter: &mut Adapter = unsafe {
    &mut *(iflib_get_softc(iflib_ptr) as *mut Adapter)
};
```

Listing 5.4: Example of dereferencing a raw pointer.

Often when there is a callback from the kernel, pointers to various objects are passed as arguments. These raw pointers has to be converted to the correct Rust structure or wrapped in a safe container. This requires either dereferencing the raw pointer which is unsafe, or a call to an unsafe function. Here one has to trust that the kernel will pass a valid pointer to the Rust code. As long as the driver is interfacing with C code, it will be difficult to eliminate all usage of raw pointers. One also might need to use raw pointers for optimization reasons in some performance critical part of the code.

5.2 Performance

The following hostnames are used for the computers in the benchmark tests:

```
A - Computer A (Hardware NIC)
B - Computer B (Bhyve NIC)
C - Computer C (Hardware NIC)
```

Listing 5.5: Computer hostnames

A list with computer specifications can be found in B.2. The device names for the e1000 driver are:

```
em - C implementation
rem - Rust implementation
```

Listing 5.6: Device names

Iperf [1] is used to transmit and receive data over a network. Computer A and Computer C are on the same machine. Computer A and Computer B are on the same local network, connected with a Gb-class router.

Command explanation:

`iperf -s` will start an Iperf server that wait for connections on port

5001.
`iperf -c <hostname>` will connect to an Iperf server on default port at host `<hostname>`.

The default is for Iperf to run for 10 seconds and then output the amount of data transferred and average bitrate. The following benchmarks are run for both em and rem.

Benchmark	Computer A	Computer B	Computer C
Hardware RX	iperf -s		iperf -c A
Hardware TX	iperf -c C		iperf -s
Bhyve RX	iperf -c B	iperf -s	
Bhyve TX	iperf -s	iperf -c A	

Table 5.1: Commands for iperf benchmarks

Network bound traffic benchmark results

First, the driver is benchmarked during transmission on hardware over a physical network. The speeds are not limited by the CPU's performance in this case.

Hardware RX

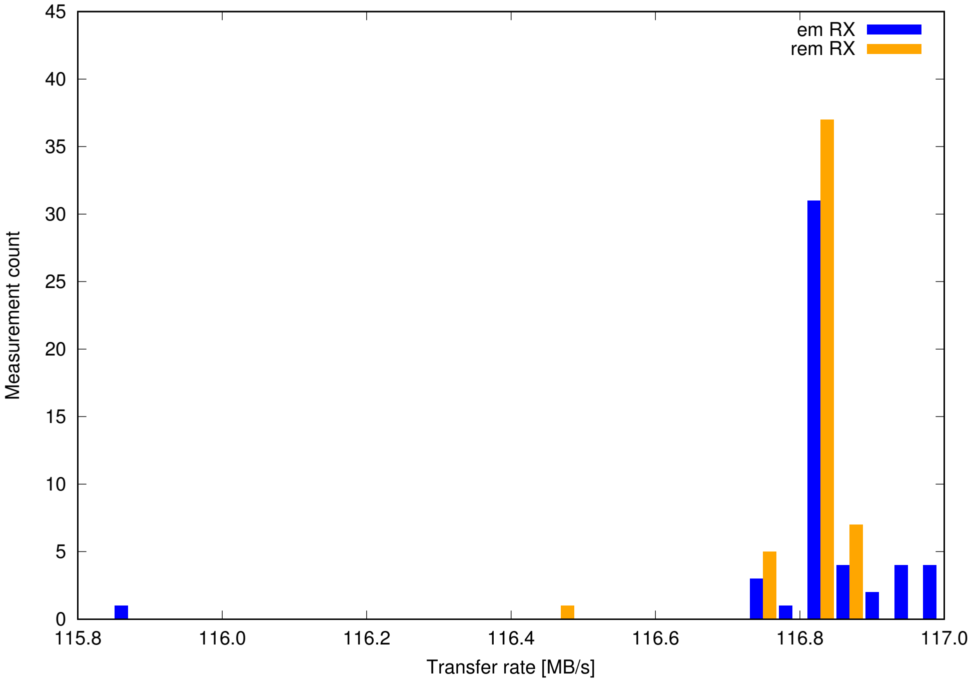

Figure 5.1: Hardware RX graph. Average bitrate for em (C) and rem (Rust) drivers receiving traffic on I219-LM hardware.

[MB/s]	Min	Max	Mean	Median
em	115.8	117.0	116.8	116.8
rem	116.4	116.9	116.8	116.8

Table 5.2: Hardware RX statistics

In this benchmark, Iperf was run a total of 50 times, each run duration is 10 seconds. Every 10th run, both machines were rebooted. Transmitting the traffic is another computer on the same local network, running stock FreeBSD 12 with native network driver.

Both drivers show a dip in bitrate, em more than rem but it is unclear why this is. It could be the result of some other network activity, caching or otherwise random event. The important result is that the behavior is the same for both implementations.

Hardware TX

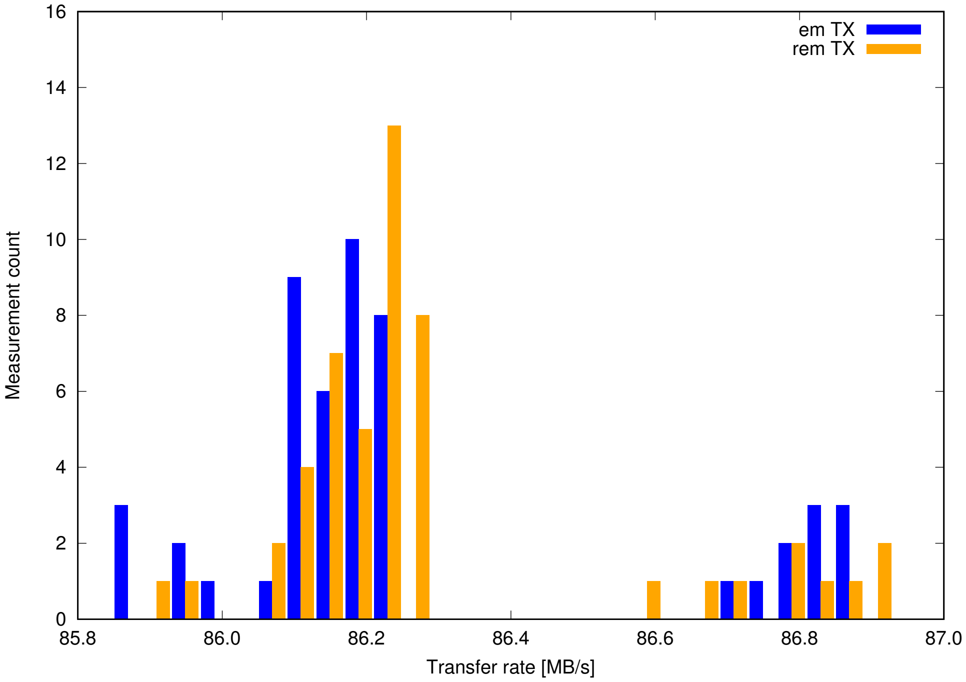

Figure 5.2: Hardware TX graph. Average bitrate for em (C) and rem (Rust) drivers transmitting traffic on I219-LM hardware.

[MB/s]	Min	Max	Mean	Median
em	85.85	86.87	86.25	86.17
rem	85.92	86.90	86.28	86.21

Table 5.3: Hardware TX statistics

In this benchmark, Iperf was run a total of 50 times, each run duration is 10 seconds. Every 10th run, both machines were rebooted. Receiving the traffic is another computer on the same local network, running stock FreeBSD 12 with native network driver.

Both drivers show an interesting spread with a gap but it is unclear as to why this is. It could be a property of the router or the receiving computer as well as the local computer of network card. What is important here as well is that both implementations perform similarly. It should be noted that transmit rate is slower than receive rate and the reason for this is unclear. Since it is the same for both implementations, this difference is ignored.

CPU bound traffic benchmark results

In the previous section the transfer rate over a physical network was measured. The transfer rate is limited by the network and the CPU is not fully utilized. To examine how the Rust implementation compares to the C implementation during 100% CPU utilization, network transfer between the host and the bhyve virtual machine is measured.

Bhyve RX

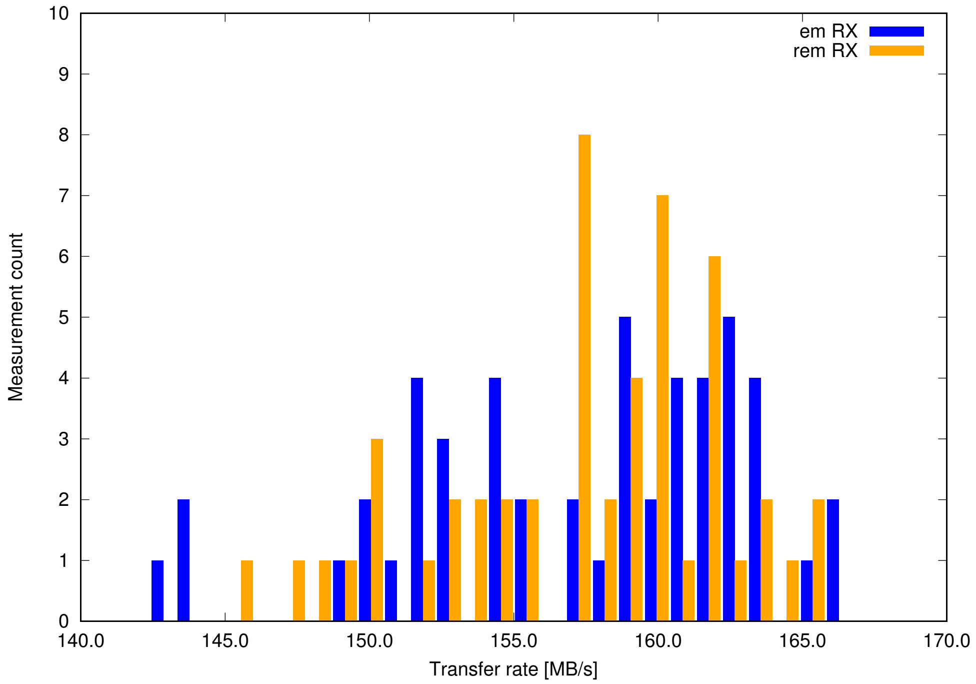

Figure 5.3: Bhyve RX graph. Average bitrate for em (C) and rem (Rust) drivers receiving traffic in Bhyve virtual machine.

[MB/s]	Min	Max	Mean	Median
em	142.5	166.2	157.2	158.9
rem	145.5	165.3	157.2	158.0

Table 5.4: Bhyve RX statistics

In this benchmark, Iperf was run a total of 50 times, each run duration 10 seconds. Receiving the traffic is the Bhyve virtual machine and transmitting the traffic is the Bhyve host. The Rust implementation performs similar to the C implementation.

Bhyve TX

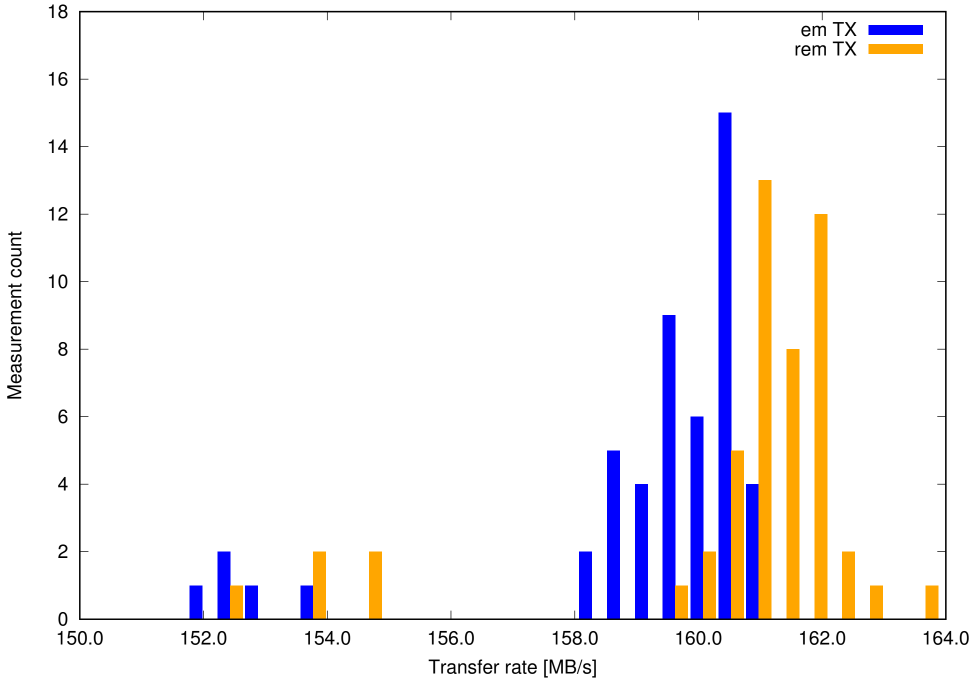

Figure 5.4: Bhyve TX graph. Average bitrate for em (C) and rem (Rust) drivers transmitting traffic in Bhyve virtual machine.

[MB/s]	Min	Max	Mean	Median
em	151.8	161.0	159.1	159.7
rem	152.2	163.5	160.5	161.0

Table 5.5: Bhyve TX statistics

In this benchmark, Iperf was run a total of 50 times, each run duration 10 seconds. Transmitting the traffic is the Bhyve virtual machine and receiving is the Bhyve host. In the beginning of each test run the bitrate was a bit slower which explains the lower outliers. This could be due to caching in the CPU. The Rust implementation performs a little bit better than the C implementation but it is not statistically significant. It is unclear why the Rust implementation performs better.

CPU benchmark results

To breakdown and analyze further, the CPU utilization of kernel and userland threads are measured during peak transmission from the Bhyve virtual machine to the host.

The following graphs show the output of the command `top -bzSHCP`, run 10 times with 1 second between each time while Iperf is transmitting traffic from Bhyve virtual machine to the host. Iperf timeout was extended to 2 minutes and the benchmarks start 1 minute after the transferred is started to allow the CPU usages to stabilize[2].

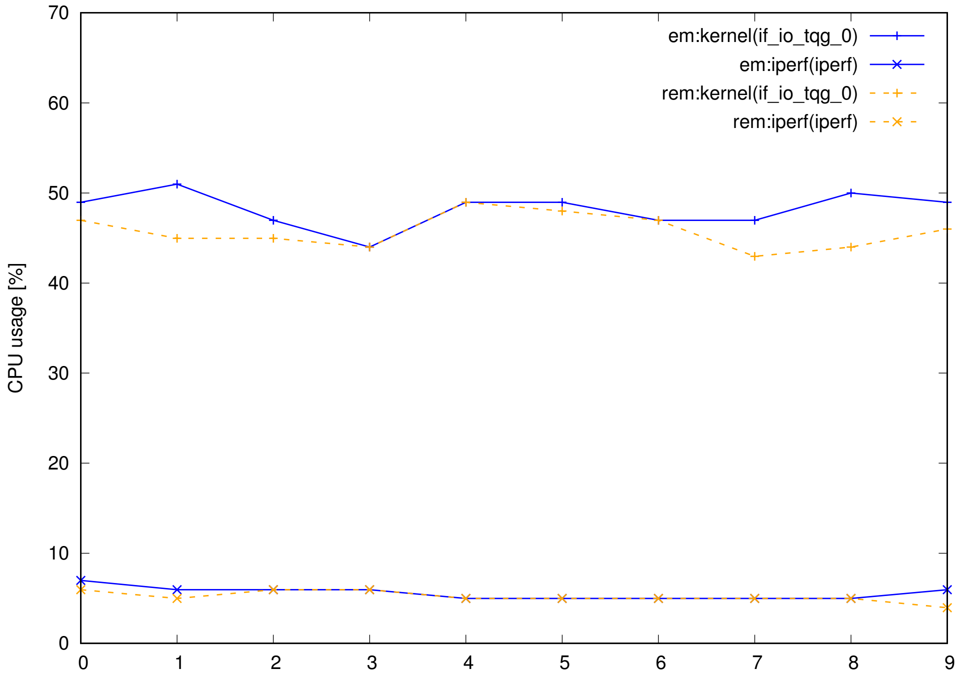

Figure 5.5: Bhyve TX - Bhyve virtual machine. A measurement of CPU usage by threads in Bhyve.

The graph shows the two threads in the virtual machine using any significant amount of CPU resources. As can be seen, there is no performance penalty in the Rust implementation.

[2] It takes about 1 minute until the e82545-tx thread reaches 100% CPU utilization

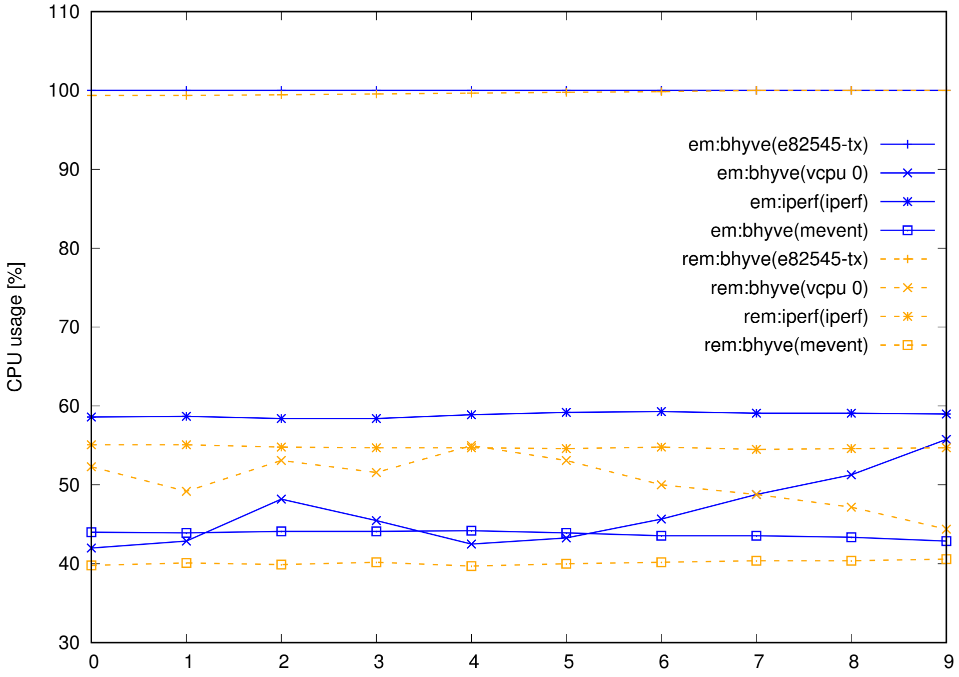

Figure 5.6: Bhyve TX - Bhyve host. A measurement of CPU usage by threads in the host.

The graph shows all threads in the Bhyve host using any significant CPU resources. The threads “mevent” and “iperf” show opposite higher load in C vs. Rust. The reason for this is unclear. The “vcpu 0” thread show mostly higher load in the Rust implementation. It could be speculated that this is the reason for higher transfer rate in Rust as shown in graph 5.4.

Chapter 6

Conclusions

In the first chapter two goals were introduced. Neither of these goals have any hard limits or fixed boundaries where they fail or succeed but rather a conclusion should be drawn from looking at the results as a whole.

6.1 Safeness

As we could see in the analysis of the unsafe code, the current state of the implementation still depends on a lot of unsafe code, more than desired actually. However, let's revisit the numbers.

- 56 calls to C functions.
- 43 calls to unsafe Rust functions.
- 73 access to unions.
- 37 dereferencing raw pointers.

Some comments and conclusions regarding this results are:

- Calls to C functions, which are always unsafe, can be reduced to zero if they are wrapped in a safe interface. A safe interface would need to include proper assertions and bound checks.
- Calls to unsafe Rust functions are unavoidable when dealing with raw pointers that need casting to other types. However, as with C functions, these can also be moved to the interface layer where they are wrapped and made safe.

- For the e1000 driver, no union is shared between the kernel and the device driver. All unions are internal to the driver and can easily be changed to something that is safe in Rust, like Enumerations.
- Dereferencing raw pointers is basically unavoidable when interfacing with C code. Because raw pointers are often arguments in callback functions it is difficult to move this code to an interface layer. Unless some very fancy safe interface is implemented, one needs to dereference raw pointers in the driver code. However, if we can trust the kernel to hand valid pointers to the Rust code, these could be classified as safe.

As seen above, with a bit more work, we can eliminate almost all use of unsafe code, with the only remaining being dereferencing raw pointers. If IfLib was ported to Rust we would not need those callbacks from C code and could probably write a network device driver completely in safe Rust code. We can not however, escape the use of unsafe code in our interface layer. The limitation here is that we have to trust the kernel and the raw pointers it manages. Internally, Rust also uses unsafe code which is wrapped in a safe interface. Similarly we need to trust that our internal unsafe code is safe if we are to wrap it in a safe interface.

It should be noted that while even unsafe Rust is safer than C, being able to write a kernel device driver completely in safe Rust is a big selling point for the politics regarding introducing another language in the kernel.

6.2 Performance

In this project we can see no performace penalty at all for using Rust instead of C. In fact, in the benchmark displayed in figure 5.4 we can see the Rust driver is performing a little bit better but at the same time, we noticed one of the CPU threads having higher CPU utilization for the Rust driver. No conclusion can be drawn from this except that maybe the Rust compiler could optimize the code better, which resulted in higher CPU utilization.

One rather important factor to include is that for the e1000 driver, IfLib, which is written in C, is doing a lot of the heavy lifting and the device driver code is mostly manipulating hardware registers and keeping an internal state. If instead another device driver like Realtek, which does not depend on IfLib and does all the heavy lifting iself, or IfLib itself, was ported to Rust maybe some difference in performance could be detected. Rust's performance compared to C is probably very dependent on the specific work load. Many of Rust's key features is about static analyzing and that does not add any runtime overhead. Boundary checks and similar extra safety code does on the other hand.

6.3 Future work

IfLib

A continuation of this project could be to port Iflib to Rust, or port some driver that does not depend on Iflib and is interfacing more with the kernel. This would provide a more interesting comparison when it comes to performance and evaluating Rust's usability in the kernel when interfacing with more complicated kernel programming interfaces.

Tracing

Dtrace and HWPMC are terrific tools for analyzing code. Another future work could be to make these tools be able to trace Rust code. This would enable profiling like measuring how much CPU time is spent in each function.

Optimizing userland applications

One interesting side effect of being able to run safe code in the kernel is that not only can we write device drivers with safe code but also move userland applications into the kernel for increased performance[1]. Some userland applications rely heavily on system calls and programmers are trying to optimize the code by reducing system calls or bundling many system calls together in one. If the Rust compiler can guarantee that the code is safe and that it will not do anything during runtime that is not allowed, performance benefits can be made by moving userland applications into the kernel.

[1] See related work section for example

Bibliography

[1] Wikipedia contributors. *Iperf — Wikipedia, The Free Encyclopedia*. [Online; accessed 25-May-2018]. 2018. URL: https://en.wikipedia.org/wiki/Iperf.

[2] The FreeBSD Documentation Project. *FreeBSD Handbook*. [Online; accessed 13-May-2018]. 2018. URL: https://www.freebsd.org/doc/handbook/nutshell.html.

[3] Wikipedia contributors. *Northeast blackout of 2003 — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-January-2018]. 2003. URL: https://en.wikipedia.org/wiki/Northeast_blackout_of_2003.

[4] Wikipedia contributors. *Therac-25 — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-January-2018]. 1985. URL: https://en.wikipedia.org/wiki/Therac-25.

[5] The Rust Team. *The Rust Programming Language*. [Online; accessed 29-January-2018]. 2018. URL: https://www.rust-lang.org.

[6] Amit Levy et al. "The Case for Writing a Kernel in Rust". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: ACM, 2017, 1:1–1:7. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124717. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/3124680.3124717.

[7] Redox Developers. *Redox Operating System*. [Online; accessed 29-January-2018]. 2018. URL: https://redox-os.org/.

[8] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2nd. 2018. ISBN: 9781593278281.

[9] Michael Hicks et al. “Experience with Safe Manual Memory-management in Cyclone”. In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM ’04. Vancouver, BC, Canada: ACM, 2004, pp. 73–84. ISBN: 1-58113-945-4. DOI: 10.1145/1029873.1029883. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1029873.1029883.

[10] Dave Clarke and Tobias Wrigstad. “External uniqueness is unique enough”. In: *In European Conference for Object-Oriented Programming (ECOOP*. Springer-Verlag, 2003, pp. 176–200.

[11] Arch D. Robison and Ralph E. Johnson. “Three Layer Cake for Shared-memory Programming”. In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ParaPLoP ’10. Carefree, Arizona, USA: ACM, 2010, 5:1–5:8. ISBN: 978-1-4503-0127-5. DOI: 10.1145/1953611.1953616. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1953611.1953616.

[12] Galen C. Hunt and James R. Larus. “Singularity: Rethinking the Software Stack”. In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424. URL: http://doi.acm.org/10.1145/1243418.1243424.

[13] Manuel F¨ahndrich et al. “Language Support for Fast and Reliable Message-based Communication in Singularity OS”. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys ’06. Leuven, Belgium: ACM, 2006, pp. 177–190. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217953. URL: http://doi.acm.org/10.1145/1217935.1217953.

[14] Abhiram Balasubramanian et al. “System Programming in Rust: Beyond Safety”. In: *SIGOPS Oper. Syst. Rev.* 51.1 (Sept. 2017), pp. 94–99. ISSN: 0163-5980. DOI: 10.1145/3139645.3139660. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/3139645.3139660.

[15] J. N. Herder et al. “Fault isolation for device drivers”. In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 2009, pp. 33–42. DOI: 10.1109/DSN.2009.5270357.

[16] Amit Levy et al. "Ownership is Theft: Experiences Building an Embedded OS in Rust". In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. PLOS '15. Monterey, California: ACM, 2015, pp. 21–26. ISBN: 978-1-4503-3942-1. DOI: 10.1145/2818302.2818306. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/2818302.2818306.

[17] Robert Clipsham (c. *Safe, Correct, and Fast Low-Level Networking*. 2015.

[18] Luigi Rizzo and Matteo Landi. "Netmap: Memory Mapped Access to Network Devices". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 422–423. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018500. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/2018436.2018500.

[19] H. Bos and B. Samwel. "Safe kernel programming in the OKE". In: *2002 IEEE Open Architectures and Network Programming Proceedings. OPENARCH 2002 (Cat. No.02EX571)*. 2002, pp. 141–152. DOI: 10.1109/OPNARC.2002.1019235.

[20] Greg Morrisett et al. "TALx86: A Realistic Typed Assembly Language". In: *In Second Workshop on Compiler Support for System Software*. 1999, pp. 25–35.

[21] Toshiyuki Maeda. *Safe execution of user programs in kernel mode using typed assembly language*. Tech. rep. 2002.

[22] E. Zadok et al. "Efficient and safe execution of user-level code in the kernel". In: *19th IEEE International Parallel and Distributed Processing Symposium*. 2005, 8 pp.–. DOI: 10.1109/IPDPS.2005.189.

[23] H. Fujita et al. "P-Bus: Programming Interface Layer for Safe OS Kernel Extensions". In: *2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*. 2010, pp. 235–236. DOI: 10.1109/PRDC.2010.31.

[24] Haris Volos et al. "xCalls: Safe I/O in Memory Transactions". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 247–260. ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519093. URL: http://doi.acm.org.focus.lib.kth.se/10.1145/1519065.1519093.

[25] FreeBSD contributors. *bhyve, the BSD Hypervisor*. [Online; accessed 15-May-2018]. 2018. URL: `https://wiki.freebsd.org/bhyve`.

[26] The FreeBSD Documentation Project. *FreeBSD Handbook*. [Online; accessed 22-Jun-2018]. 2018. URL: `https://www.freebsd.org/`.

[27] The FreeBSD Documentation Project. *FreeBSD Ports*. [Online; accessed 22-Jun-2018]. 2018. URL: `https://www.freebsd.org/ports`.

[28] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: `http://dl.acm.org/citation.cfm?id=977395.977673`.

Appendix A

Development process

This appendix is a documentation of the development process. It shows some limitations of the tools used and highlights some of the things needed to be handled in future work that was not done in this thesis.

A.1 Kernel bindings

Bindgen is a tool that helps with generating C bindings for Rust. While it can generate code for complex things like unions and bit fields, at time of writing there are a few places where it falls short.

These are

- **Macro definitions** - Constants are all generated in decimal form. Usually hardware registers, bitmaps, bitmasks, etc are declared in hexadecimal form. An option for this in bindgen would make porting easier.
- **Paths** - For generated code like debug printing, the std::fmt path is hardcoded in and has to changed after generating bindings. Here this is done using sed in the Makefile.
- **Function style macros** - As of bindgen 0.33, function style macros can not be converted. The e1000 driver depend heavily on these to calculate memory offsets to hardware registers. Manually transcribing these can be a cause of hard to find bugs.

A.2 Mutexes

For this project the spin crate was used to provide Mutex and RWLock. This create supports the no_std feature and in this case the mutex is implemented using a simple spin lock. Compared to userland, mutexes in the kernel have different options and while it works for simple functions, simply plugging the spin crate into RustKPI is not a sufficient solution. A proper kernel sync crate would have to be implemented.

A.3 Malloc

Kernel malloc takes an argument which userland malloc does not. Among other settings, this argument decide if malloc should wait until enough memory is available so that the allocation can be made, or if it should return immediately with an error if not enough free memory is available. In RustKPI we implement an allocator which calls kernel malloc and Rust's crate liballoc uses our allocator for all allocations. In this way Rust is doing all the heavy lifting for us. However, with liballoc there is no way to pass an extra flag when allocating memory using Box::new() or similar. To enable the use of extra malloc arguments one would have to patch Rust's liballoc and allocator API.

A.4 Concatenate identifiers

C pre-processor macros in the kernel often concatenate names to form new, unique names for structures and variables. At time of writing Rust did not support this. While it might be possible to work around this in Rust, a compiler plugin called interpolate_idents was used for this project to provide the same capability for Rust macros.

A.5 Automatic code conversion

Several hard to find bugs in the Rust code was caused by error made during manual transcribing from C. While it's difficult to automatically convert all C code to Rust, a tool that could convert the bulk of it would surely reduces bugs caused by human error.

Appendix B

Listings

B.1 Hello world kernel module in Rust

```
#![feature(const_fn,plugin,used,global_asm,rustc_private)]
#![plugin(interpolate_idents)]
#![no_std]

#[macro_use]
extern crate kernel;

use kernel::sys::ModEventType;
use kernel::sys::module_sys::module_t;
use kernel::sys::module_sys::moduledata_t;
use kernel::sys::kernel_sys::sysinit_sub_id;
use kernel::sys::kernel_sys::sysinit_elem_order;
use kernel::sys::raw::c_int;
use kernel::sys::raw::c_void;

#[derive(Debug)]
struct A(i32);
impl kernel::ops::Drop for A {
    fn drop(&mut self) {
        println!("rustkpi-hello:␣A::drop()␣{:?}", self);
    }
}
/*
 * The kernel expects a C function for the event callback.
 */
pub extern "C" fn module_event(_module: module_t, event:
   c_int, _arg: *mut c_void) -> c_int {
    match ModEventType::from(event) {
        ModEventType::Load => {
            println!("rustkpi-hello:␣Got␣kernel␣module␣event:
               ␣LOAD");
            /*
             * Create a vector with heap allocated storage.
             * Control that it is released when the
             * variable 'v' goes out of scope by watching
             * the output from A's drop() function.
             */
            let mut v = vec![A(0), A(1), A(2)];
            println!("rustkpi-hello:␣Vector␣is:␣{:?}", v);
        }
        ModEventType::Unload => {
            println!("rustkpi-hello:␣Got␣kernel␣module␣event:
               ␣UNLOAD");
        }
        ModEventType::Quiesce => {}
        ModEventType::Shutdown => {}
        ModEventType::Unknown => {}
    }
    0
}

pub static MODULE_DATA: moduledata_t = moduledata_t {
    name: b"rustkpi_hello\0" as *const u8 as *const i8,
    evhand: Some(module_event),
    priv_: 0 as *mut c_void,
};

/* These macros require interpolate_idents compiler plugin */
declare_module!(
    rustkpi_hello,
    MODULE_DATA,
    sysinit_sub_id::SI_SUB_DRIVERS,
    sysinit_elem_order::SI_ORDER_MIDDLE
);
module_depend!(rustkpi_hello, rustkpi, 1, 1, 1);
```

Listing B.1: Hello World kernel module in Rust.

B.2 Hardware setup

```
Computer A
Model: Dell Latitude 7270
CPU: Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz
NIC (hardware): Ethernet Connection I219-LM class=0x020000
   card=0x06db1028 chip=0x156f8086 rev=0x21 hdr=0x00
```

```
Computer B
Model: bhyve virtual machine (running in Computer A)
CPU: Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz
NIC (emulated): 82545EM Gigabit Ethernet Controller class=0
    x020000 card=0x10088086 chip=0x100f8086 rev=0x00 hdr=0
    x00

Computer C
Model: Dell Latitude 7450
CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
NIC (hardware): Ethernet Connection (3) I218-LM class=0
    x020000 card=0x062e1028 chip=0x15a28086 rev=0x03 hdr=0
    x00
```

Listing B.2: Computer setup

B.3 LLVM target configuration

```
{
    "llvm-target": "x86_64-unknown-freebsd",
    "linker-flavor": "gcc",
    "target-endian": "little",
    "target-pointer-width": "64",
    "target-c-int-width": "32",
    "os": "none",
    "arch": "x86_64",
    "data-layout": "e-m:e-i64:64-f80:128-n8:16:32:64-S128",
    "pre-link-args": [ "-m64" ],
    "cpu": "x86-64",
    "features": "+soft-float,-mmx,-sse,-sse2,-sse3,-ssse3,-
       sse4.1,-sse4.2,-3dnow,-3dnowa,-avx,-avx2",
    "disable-redzone":         true,
    "custom-unwind-resume":    true,
    "eliminate-frame-pointer": false,
    "stack-probes":            true,
    "linker-is-gnu":           true,
    "no-compiler-rt":          true,
    "archive-format":          "gnu",
    "code-model":              "kernel",
    "relocation-model":        "static"
}
```

Listing B.3: A LLVM target file for compiling Rust to FreeBSD kernel target.

B.4 e1000 C files

```
 1.6k LICENSE
  12k README
  41k e1000_80003es2lan.c
 4.3k e1000_80003es2lan.h
  20k e1000_82540.c
  36k e1000_82541.c
 3.7k e1000_82541.h
  16k e1000_82542.c
  45k e1000_82543.c
 2.5k e1000_82543.h
  54k e1000_82571.c
 2.8k e1000_82571.h
 101k e1000_82575.c
  21k e1000_82575.h
  38k e1000_api.c
 7.9k e1000_api.h
  66k e1000_defines.h
  27k e1000_hw.h
  22k e1000_i210.c
 4.3k e1000_i210.h
 176k e1000_ich8lan.c
  14k e1000_ich8lan.h
  71k e1000_mac.c
 5.1k e1000_mac.h
  16k e1000_manage.c
 4.2k e1000_manage.h
  20k e1000_mbx.c
 5.4k e1000_mbx.h
  32k e1000_nvm.c
 3.7k e1000_nvm.h
 3.2k e1000_osdep.c
 9.1k e1000_osdep.h
 118k e1000_phy.c
  14k e1000_phy.h
  38k e1000_regs.h
  17k e1000_vf.c
 9.0k e1000_vf.h
  24k em_txrx.c
 145k if_em.c
  17k if_em.h
  17k igb_txrx.c
```

Listing B.4: e1000 C implementation found in sys/dev/e1000 in the FreeBSD source code.