Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Writing Network Drivers in C#

Maximilian Stadlmeier

Abstract

User space network drivers on Linux are often used in production environments to improve the performance of network-heavy applications. However, their inner workings are not clear to most programmers who use them. ixy aims to change this by providing a small educational user space network driver, which is gives a good overview of how these drivers work, using only 1000 lines of C code.

While the language C is a good common denominator, which many developers are familiar with, its syntax is often much more difficult to read than that of more modern languages and makes the driver seem more complex than it actually is.

For this thesis I created a C# version of ixy, named ixy.cs, which utilizes the more modern syntax and additional safety of the C# programming language in order to make user space network driver development even more accessible. The viability of C# for driver programming will be analyzed and its advantages and disadvantages will be discussed.

Contents

Chapter 1

Introduction

1.1 Kernel modules and user space drivers

Drivers for Linux systems are generally written as kernel modules. These modules run in kernel space and therefore have certain priviliges, such as reacting to hardware interrupts. However, there are drawbacks to running in kernel space. There is a computational overhead for each kernel call due to the necessity to switch between kernel and user mode, so frequent calls to kernel functions can noticeably slow down the software. Many tools which help with the development and testing of software, such as Valgrind are not available in kernel space and development is generally much more cumbersome. Furthermore, kernel modules can only be written in a subset of the C programming language.

The reliance on C and lack of easily usable diagnostic tools make it necessary to be exceedingly careful about memory leaks, stack overflows, segfaults and other typical C errors, which can have much more severe consequences in kernel space, where there are fewer safeguards than in user space. As a result, a carelessly written or poorly tested kernel space driver may easily crash the system, overwrite data from other applications or corrupt the file system.

User space drivers mitigate most of these problems and provide a simpler, more flexible alternative to kernel modules. They can in principle be written in any programming language, without any restrictions on which frameworks or functions are usable because they run in user space, just like any other application. The same profiling, diagnostics and debugging tools can be used for user space driver development as for application development, which not only makes the development process much more convenient and cheaper but also tends to result in better, safer software. When bugs and errors do occur, they have less severe consequences, as the user space driver cannot write to any protected address space, such as kernel memory or memory of other applications. Furthermore, due to the ability to choose any programming language to write the driver, many issues, such as memory leaks, that C developers frequently face, may be avoided.

A user space driver can also perform significantly better than its kernel space equivalent. With kernel space drivers, the processor constantly has to switch between user mode on ring 3 and privileged kernel mode on ring 1, which is associated with a non-negligible computational overhead. This overhead is particularly noticeable if data frequently has to be copied between user and kernel space, which is the case in network drivers. A user space driver does not have this overhead, as the entire driver runs entirely in user space. Intel claims that their DPDK driver increases performance by a factor of ten and in their benchmarks a forwarder written with their driver can operate at very close to line rate [7]

1.2 ixy and ixy.cs

Due to their speed, user space drivers are often plugged into existing applications in order to increase performance. Even in production environments, programmers often do not know how the user space drivers speeding up their applications actually work or how they increase performance.

In order to change this, the ixy user space network driver was developed [1]. ixy is a user space driver for NIC devices, which aims to make user space driver development accessible to everyone. The driver is written with simplicity as its primary design goal and is implemented in only around 1000 lines of C code. ixy aims to make it clear how user space network drivers like DPDK work while balancing simplicity and efficiency.

The language C was used for the development of ixy because it is a suitable common denominator, which most developers are at least somewhat familiar with. However, while C is a widely known language, it is generally considered to be rather difficult to read and write, provides few abstractions and in general makes driver development seem more complex than it actually is. Programmers not commonly involved in systems- or driver development, in particular, may have a difficult time comprehending ixy's source code in C.

In order to make ixy more accessible, I have created a port of ixy called ixy.cs, which is written in the C# programming language. C# is a memory and type safe language offering modern syntax and a great amount of convenience features as well as garbage collection and managed memory. Despite its higher-level nature, C# still allows access to low level constructs such as pointers or memory mappings, making it an ideal choice of language for writing an educational user space network driver. In this thesis I analyzed the viability of C# as a programming language for systems programming in general and real-world user space network driver development in particular, which advantages it provides and which challenges it brings compared to C.

Chapter 2

On The Coice Of Language

2.1 C# Overview

C# is a strongly typed, imperative, object-oriented programming language developed by Microsoft. The language's syntax is quite similar to that of C, C++ and other "C-like" languages such as Java. It compiles to a platform-independent "Common Intermediate Language", which is then compiled Just in Time to machine code by a runtime. For the C# port of the ixy driver I used version 2.1 of the modern cross-platform and open source runtime .NET Core.

C# applications use fully managed, garbage collected memory by default but the language also allows for manual allocation, reading and writing of unmanaged memory in a so called "unsafe context". C# comes with the .NET framework, which provides an enormous amount of abstractions, interfaces, data structures, algorithms and helpers for a wide variety of tasks. The language supports functional programming to a certain degree through the use of first class functions and supports lambda expressions. The latter, combined with the built-in query language LINQ, can be used to perform operations on data collections in a very expressive and easily readable way. C# provides all the typical object-oriented features, such as single inheritance, interfaces and polymorphism and supports concurrency and asynchronicity.

The actual development process when writing a network driver in C# is very close to how a C version would be written. In most places, C# syntax looks similar to C syntax. However, C# is easier to read for someone who does not have C experience. Its object-oriented style makes the source logic simpler to understand and by avoiding pointers, manual memory allocation and other typical C operations where possible, the code becomes a lot more clear.

Another point in C#'s favor is its similarity to other languages. Java code, for instance, is often nearly identical to C# code and other languages, such as JavaScript, also resemble C# more closely than C. Thus, C# aids ixy in its goal of being accessible to as many programmers as possible.

2.2 Viability of C# on Linux

On first glance it might seem an unusual choice to use a Microsoft-designed programming language on a Linux-based software project. I would like to address this and explain why I believe C# to be a suitable choice of language for writing a network driver on Linux.

In the past, portability was a tremendous problem for C#. Its use on any non-Microsoft platform was extremely limited due to the fact that the .NET framework, the runtime and the compiler were closed source and developed for Windows without Linux or Mac compatibility in mind. However, there has been constant progress in porting C# along with its dependencies and tooling to other platforms. The Mono project started in 2004, not long after the initial release of C# and has been used in many projects and on various platforms. Its goal is to provide an implementation of C# and the .NET framework which is as identical to Microsoft's version as possible, while not being restricted to Microsoft platforms. While the implementation started out as quite incomplete, it has been actively improved since its inception and now implements the vast majority of the Microsoft version's functionality. As a consequence, C# has been becoming a viable programming language on Linux in recent years. In 2016 Microsoft released .NET Core, a complete open source rewrite of the .NET framework and runtime, which natively supports Windows, Linux and Mac. Its API is mostly identical to the original .NET API, but modernized in some areas. With this development, C# is actually one of the more portable programming languages and the implementation of the language and framework on Linux is complete, without the need for developers to write any platform specific code. Today, C# and .NET treat Linux as a first class citizen with development being identical on all platforms, aside from some development tools which are not yet cross-platform.

Perhaps partly due to these efforts, C# now belongs to the most popular and widely used programming languages. In the 2018 StackOverflow survey [5], 35% of professional developers reported to have worked with C# at some point. The language is being used for almost all types of software. On the web, ASP.NET is consistently among the most widely used frameworks for server-side development, especially in enterprise environments. C# mobile development is possible for both Android and iOS with the Xamarin framework and desktop applications can be built with WinForms or WPF for Windows or cross-platform alternatives. Another area where C# is very commonly used is game development with the most prominent C# using software suite being the Unity 3D game engine. While C# is very rarely used for systems or driver programming, it provides all the functionality and performance required to develop this kind of software, as my port of ixy shows. Furthermore, the choice of C# becomes even more natural when one considers that one of the goals of ixy is making driver code accessible to application developers who might not otherwise have had the desire or the expertise to study and comprehend thousands of lines of C code.

Chapter 3

Related Work

As mentioned in the previous section, C# is not commonly used for any sort of low level systems or driver programming. C still dominates this space, despite being used less and less in many other areas of development. There are, however, some projects which do use C# for a wide variety of system level tasks. Most notably, Singularity, Cosmos and SharpOS, three operating systems written almost entirely in C# or C# dialects. The most well-known and notable of these is Singularity, which also implements a network driver.

3.1 Singularity

Singularity [4] was an experimental operating system, written and designed by Microsoft in the Sing# programming language - a dialect of C#, which was developed specifically for use in this project. Sing# added some low level constructs and various meta-programming functionality to the existing dialect Spec#. Spec# in turn added constraints, preconditions, postconditions and invariants to the C# programming language. The additions of Spec# and Sing# make C# an even safer and more predictable programming language. This was of very high importance to the project, as Singularity was designed with a focus on very high dependability, stability and safety. The choice of C# is at the core of the system's design and safety is mostly implemented through language constructs, rather than traditional and more complex safety features of Unix systems, for example. All applications on Singularity, with the exception of so called "trusted" software, is guaranteed to be memory and type safe by the language runtime. Required system resources, such as access to devices have to be specified in the application's manifest and granted by the operating system during the installation process. Singularity aims to make issues such as memory leaks as well as many common security issues obsolete.

Apart from a small part of the microkernel which is written in C, the entire system is implemented in Sing#. This includes the filesystem, basic operating services and drivers, like a network driver for Intel 8254x PCI Ethernet Cards. The driver works very similarly to user space drivers like ixy and ixy.cs. Aside from the scope and functionality being somewhat broader, the biggest difference is that Singularity's network driver makes use of hardware interrupts as a kernel space driver on Linux would. Since the driver has more privileges in Singularity than a user-space driver on Linux, this is not particularly challenging to implement for a Singularity driver. After enabling interrupts through certain register flags, the driver's main worker waits for a hardware interrupt. When one is received, the cause of the interrupt is determined by reading an NIC register and the interrupt is processed accordingly. Another area where the Singularity network driver has an advantage over ixy or other Linux user space drivers is DMA memory allocation. Due to the way Singularity is designed, an application can, after declaring which resources it requires in which quantity, very easily retrieve these resources. This leads to the driver being able to allocate physically coherent, non-movable memory with just one function call. The physical address of this memory area can also be retrieved with one simple call to one of Singularity's API functions. Similarly, writing to device registers is somewhat easier because instead of memory-mapping a file, the driver can just directly write to IO memory. This is done in a type- and memory-safe C# manner without the use of pointers.

Aside from the use of hardware interrupts, one of the biggest architectural differences is that the Singularity driver keeps its own version of the NIC's ring buffer. This includes the ring capacity, the head and tail index and an array of map entries, mapping the physical packet addresses to the virtual address space. The state of this copied ring buffer is updated from data that the NIC provides. netstat similarly stores a copy of the ring buffer, with the difference being that Singularity does not expose the NIC ring buffer to other applications and only uses it internally. While keeping a copy of the ring buffer can be done in user space drivers as well, this is not done in ixy because it adds unnecessary complexity and bulk to the code.

It is very interesting to observe that the Singularity driver does not have any classes or structs for the NIC descriptors. Instead, ring buffers just provide two functions for accessing descriptors: ReadDescriptor and WriteDescriptor. These functions are used to respectively read and write the physical packet address and some control bits from or to the descriptor with a given index. Object oriented programming principles would dictate that for something like a descriptor, which is clearly a logical object, containing data and references, a class or struct should be created. A likely reason for the driver's lack of these objects is that when transmitting or receiving data, a very large amount of descriptors would have to be allocated, updated and eventually cleaned up. This is very costly in C# where excessive object allocation and disposal can cause bottlenecks due to the garbage collector. This subject will be discussed further in the performance analysis section of this thesis.

It is worth noting that the driver manages to work entirely without any C code or unsafe C# code. All reading and writing is done through safe Sing# functions and objects provided by the language and Singularity's memory API.

This driver and Singularity in general are noteworthy, because they show that systems programming in C# or a language derived from it is not only possible, but has the potential for creating very dependable and safe software. Furthermore, performance tests by Microsoft have shown that in virtually every respect, Singularity's performance is equal or even better than that of traditional operating systems written in lower level languages, which proves that using a safe, managed language does not necessarily negatively impact performance. It should also be kept in mind, however, that Singularity's network driver benefits from being built for an operating system which is specifically designed to work well with Sing#. This causes some aspects of the driver, in particular memory allocation and handling of the DMA memory, to look easier to implement than they are on traditional operating systems like Linux.

Chapter 4

Driver Architecture

The source code for ixy.cs can be found under the following GitHub page: `https://github.com/ixy-languages/ixy.cs` . Unless specified otherwise, any code that is mentioned refers to the commit `484485b` in the `master` branch of this repository. Instructions for installation and usage can be found in the `README.md` file.

The C# port of ixy is mostly identical in design to the original C version, which itself is designed to closely resemble DPDK. The precise processes for initialization as well as receiving, transmitting and batching packets are described in the Intel 82599 datasheet [3], which was vital during the development of ixy and ixy.cs. The primary design goal of ixy is simplicity. The driver should be easy to read and extend. This means that in some cases functionality is limited in order to minimize complexity. Missing functionality includes various offloading features, which could be implemented in a user space driver, but have been cut. The driver APIs layer of abstraction is also fairly limited and not designed to be as user-friendly as possible, but rather to make it clear to the application programmer how exactly the driver works. However, a higher level API could easily be built on top of ixy and most missing functionality can be added at the cost of increasing the driver's complexity.

4.1 Driver Abstraction

The original ixy driver has some level of abstraction when it comes to device-specific driver implementations. There is the `ixy_device` struct with little more than function pointers to all the functions that the API needs to provide, such as batching, reading device statistics or retrieving the link speed. Device specific driver implementations (currently Ixgbe and VirtIO are implemented in the C version) provide another struct, such as `ixgbe_device`, which include an `ixy_device` and contain implementations for the aforementioned functions. While this setup works, it is certainly less than ideal to pass a reference to an `ixy_device` to almost every function of the various driver implementations. In C# this can be solved much more elegantly through proper inheritance and polymorphism. The application programmer directly instantiates a device class for the desired type of device. This device inherits from the `IxyDevice` class, which includes data such as the device's PCI address, exposes the required API and also offers common functionality, such as read and write functions to device registers to any specific driver implementations that inherit from this class. The result is a much more readable and more easily extendable architecture. Since ixy.cs only implements the Ixgbe driver for Intel's 82599ES NIC devices, the rest of this thesis will only examine the Ixgbe driver.

4.2 Initialization Process

In its constructor `IxgbeDevice` starts the initialization process by unbinding any currently active driver for the NIC. This is done very simply by writing to the unbind file for the NIC's pci address in the sysfs file system. The process of unbinding the current driver is one of several which require superuser privileges. Secondly, DMA has to be explicitly enabled for the device, which is also done by writing to a sysfs file. In order to perform the next configuration steps, the driver has to map the device registers into memory. C# enables memory mapping with the `MemoryMappedFile` class. `MemoryMappedViewAccessor` then provides a type and memory safe way of writing to and reading from the mapped memory. This accessor can also be restricted to a certain segment of the mapped memory, which could be useful for hardware where only some device registers should be accessible by the driver. Device registers can be accessed and written to directly, without any pointers or unsafe C# code, using nothing but built-in APIs of the .NET framework. `IxyDevice` utilizes this accessor and implements functions for reading or setting device registers.

These register management functions are used to configure the NIC for the driver. For instance, interrupts are disabled, as ixy relies on polling instead. The NICs responses can be awaited by continuously reading a certain register until a certain value has been written. This is done repeatedly throughout the initialization process, for example after performing a global device reset.

4.3 DMA Memory Allocation

After the device has been reset and the device performed some low level link negotiation, the driver prepares for receiving and sending packets in the `InitRx` and `InitTX` methods, respectively. This includes some more configuration through device registers, like enabling CRC offloading, but is also where the driver allocates memory for use in DMA.

NICs work with queues for both RX and TX. Both incoming and outgoing packets can be distributed between different queues to be processed individually or merged. These queues are filled with RX or TX descriptors which contain pointers to packet buffers as well as some metadata. During RX and TX initialization, the driver has to allocate memory for these descriptors per queue. While C# does allow directly allocating memory in safe code with the `Marshal.AllocHGlobal` function, this function did not prove to be suitable for allocating DMA memory. The reason for this is that DMA memory has to satisfy several requirements. It has to be physically contiguous, it has to remain in the same physical location at all times and it may not be swapped. Unfortunately, memory allocated by `AllocHGlobal` does not fill all these requirements.

Swapping can be disabled with the `mlock` function, which is not exposed by the .NET framework. However, even after disabling swapping, the memory's physical address can still be moved by the kernel. ixy and other user space network drivers overcome this issue by using explicitly allocated huge pages, which Linux does not currently relocate. They can be used by mounting pages in the `hugetlbfs`, which the driver can then map to memory.

As C# cannot call `mlock` or get a raw pointer from a memory mapped file, DMA memory allocation is performed in C and called with the C# P/Invoke mechanism. Fortunately, this is the only instance of the driver calling a C function and the total amount of C code is only around 30 lines. After allocating the memory through hugepages, there is still the non-trivial task of getting the physical address from the virtual address referenced by the pointer. This can be done with the pagemap in /proc/self/pagemap, which maps virtual and physical addresses of the current process. This translation is done in the `VirtToPhys` function of `MemoryHelper`.

The physical address can now be communicated to the NIC through the usual channel of registers and also saved in the queue. While configuration and the sending of commands to the NIC happens through registers, the actual data in the form of RX or TX descriptors is written to this shared memory that can be accessed by both the driver and the NIC.

4.4 Mempools and Packet Buffers

In addition to a pointer to DMA memory containing descriptors, each queue also has a mempool. Mempools are essentially stacks of packet buffers, which are data structures that are also stored in DMA memory. They are separated into a header, which is 64 bytes large and can contain metadata such as the ID of the associated mempool, the buffer's physical address, the buffer size and so on, as well as the actual packet data.

In the C version, the buffers are implemented as simple structs with each struct having a specific address in the DMA memory and a fixed memory layout. However, as C# does not provide the same guarantees and flexibility as C when it comes to the layout of data and reading creating structs from memory addresses is expensive, I chose a different way of implementing these data structures. Instances of the `PacketBuffer` class do not reside in DMA memory and do not in fact store any packet buffer data or metadata. Instead, they only keep a pointer to the real packet buffer in DMA memory. When one reads from or writes to these PacketBuffer objects, the objects read or write data from the actual packet buffer through the pointer they store. For instance, when a caller assigns a value to the Size field of a PacketBuffer, the class takes the given 32 bit integer and writes it to the buffer's virtual address with an offset of 20 bytes. This may seem cumbersome, but through the use of C# properties, these operations look exactly like any other read or write operations on member variables to the caller.

Referencing the corresponding mempool in a packet buffer provides an additional challenge. In the send and receive functions, the driver has to be able to determine which mempool a buffer belongs to. While the C version of ixy can just save a 64 byte pointer to a mempool in the buffer, the C# version cannot use direct pointers for managed classes such as Mempool. Thus, ixy.cs assigns each mempool an ID and registers them in a static collection. The PacketBuffer then contains the ID of its mempool as an integer, which can later be used to find its parent mempool.

RX and TX descriptors were initially implemented in the same way as `PacketBuffer`. In fact, directly creating a struct from a memory address is impossible for these structures in C#, as the C version and the NIC use unions for the descriptors, which do not exist in C#, so access to the fields of the descriptors in C# has to happen in the same way as with packet buffers. In a later stage of development, the RX and TX descriptor structs were removed and replaced by a function based system, as used by the Singularity network driver. Instead of instantiating objects for the descriptors, read and write functions of the queues are called with the descriptor addresses as parameters. This approach increases forwarding performance by four percent without having a big impact on the readability or simplicity of the code. A version of the driver with the original implementation for RX and TX queues can be found in the `keep-descriptors` branch of the repository.

One mempool is allocated for each queue and its `PacketBuffer` objects are pre-allocated for performance reasons which will be discussed in more detail in a later section. After the packet buffers are allocated, each of the queue's descriptors receives a pointer to one of the queue's mempool's packet buffers. Finally, the queue is enabled by writing to an NIC register. At this point, the driver waits for the link to be established and initialization is complete.

4.5 Receiving and Transmitting Packets

Receiving and transmitting happens in batches in ixy. The reason for this is that after each receive or transmit operation, the device register RDT or TDT has to be updated. As writing to a register is a comparatively expensive operation, this is only done once at the end of each batch, severely increasing throughput. The batching functions `RxBatch` and `TxBatch` can be found in `IxgbeDevice.cs`.

The actual sending and receiving works through a ring structure provided by the NIC. The queue of descriptors is accessed by both the driver and the device and the descriptors' ownership is transferred through head and tail pointers which can be accessed via NIC registers. The hardware controls the head pointer, the driver the tail. By moving the head forward, the device gives the driver another descriptor to process and by moving the tail, the driver returns a descriptor to the device. The descriptors contain a physical pointer to a packet buffer, which the device can write to or read from directly. The driver, however, has a separate table containing the packet buffer's virtual address. While the receiving function simply goes through any descriptors it has access to, reading the associated packets, the transmission function is slightly more complex due to the asynchronous nature of transmitting packets.

The packets which the driver sends to the ring are not sent immediately by the NIC. Thus `TxBatch` has to work asynchronously. In the first part of the function any previously sent packets are cleaned up and returned to the mempool. Only after freeing these packets are new descriptors written to the ring. After writing as many new packets as the ring allows, the tail pointer is updated in the NIC register, signaling the NIC to start sending out the newly written packets.

4.6 Driver API

The ixy driver is designed to be educational and favors simplicity over an extensive API. Despite this, ixy is quite simple to use, with a basic forwarding demo including benchmarking being around 70 lines of C# code (see `Forwarder.cs`).

All initialization occurs solely in the constructor of `IxgbeDevice`, minimizing the chance for user errors due to incorrect or omitted initialization steps. Receiving packets is similarly done in just one function. `IxgbeDevice.RxBatch` takes a queue index and the desired batch size and returns an array of received `PacketBuffers`. This array's size can not exceed the batch size but the array can be smaller or even empty, depending on how many packets were actually read. It is the application's responsibility to poll RxBatch at a reasonable frequency, keeping in mind that polling has implications on energy consumption.

Sending packets can also be as simple as just calling one function, provided one already has a valid array of packet buffers. A forwarding application can simply call `IxgbeDevice.TxBatch` with a queue index and the array returned by `RxBatch`. `TxBatch` will then return the number of packets written to the NIC ring. Unsent packets can be resubmitted in the next call to `TxBatch` or freed, but should not be discarded without returning them to the mempool. Transmitting packets is more complicated for a packet generator which needs to supply its own array of packet buffers. The application will have to allocate its own mempool, write data to its buffers along with their ip checksum and keep track of a sequence number. This can be done in less than 100 lines of C# code as well, as shown in the `PacketGenerator.cs` example.

Chapter 5

Performance Analysis

In order to evaluate the viability of C# as a language for network driver development, ixy.cs should be compared directly to its C equivalent. Since its features and in most parts also its architecture are equal to that of ixy, a comparison directly shows what the performance impact of choosing C# over C is. Both drivers implement a demo forwarding application, which receives packets, changes one byte in each packet's payload in order to simulate a realistic workload and then forward them bidirectionally. The packets have a payload of 60 bytes and are processed in batches of a fixed size. Both drivers are implemented without multi-threading.

The benchmarks in figure 5.1 show how both drivers' performance is affected by the batch size. Both drivers show strongly diminishing returns when the batch size is increased. While ixy performs best with a batch size of 32, ixy.cs benefits from having a higher batch size and performs 29% slower than ixy if the batch size is decreased to 16. The difference in performance between ixy and ixy.cs becomes negligible at higher batch sizes, in particular above 128.

Figure 5.2 shows the performance of ixy.cs relative to ixy with varying CPU clockrates and a constant batch size of 32. These benchmarks show that the relative performance difference between the drivers becomes smaller with increased clockrate. When throttling the CPU to 49% of its standard frequency, ixy.cs only has 75.40% as much throughput as ixy. Enabling the Intel Turbo Boost feature, which allows the CPU to accelerate to up to 3.6GHz, increases the throughput of ixy.cs to 89.20% relative to ixy.

It should be noted that one operation in particular contributes to the performance difference between the drivers. Touching each packet after it is received is meant to simulate a realistic workload - this procedure slightly increases performance for ixy while significantly slowing down ixy.cs, due to access to the packet buffers being relatively expensive for it. When this step is omitted for both drivers, the performance difference between them largely disappears, with ixy having a bi-directional throughput of 27.41 Million packets per second versus the C# version's 27.35 using a batch size of 32 and the CPU's native clockrate of 3.3 GHz.

5.1 Benchmarks

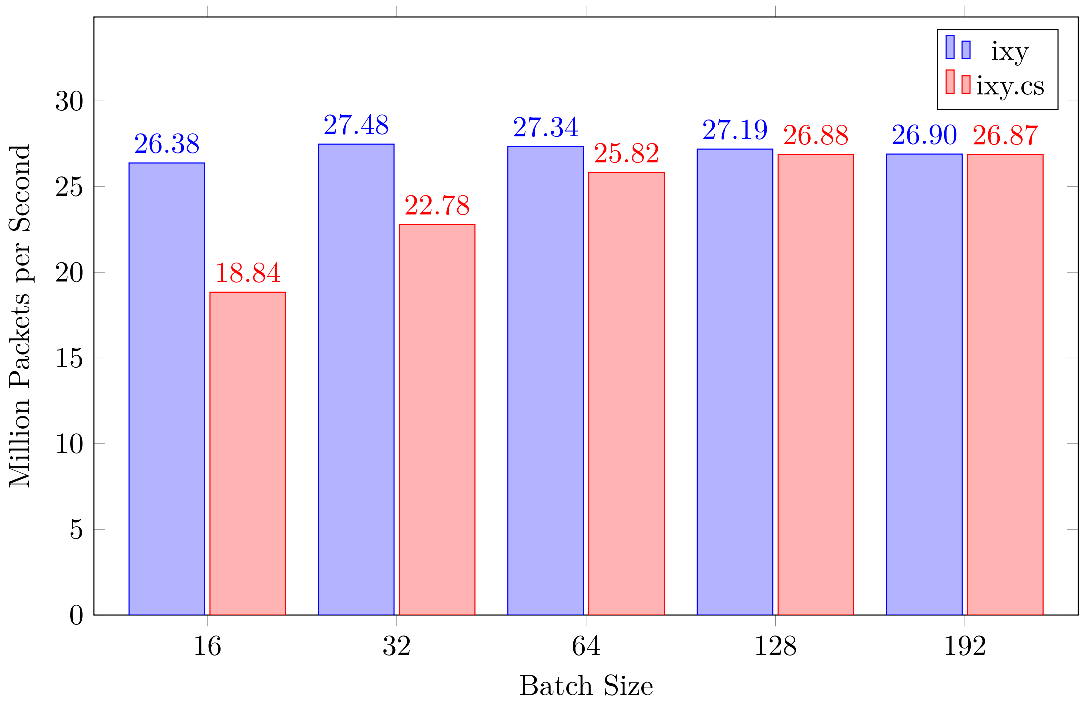

Figure 5.1: Bi-directional forwarding benchmark comparing ixy and ixy.cs with varying batch sizes on an Intel Xeon E3-1230 processor with a clockrate of 3.30 GHz

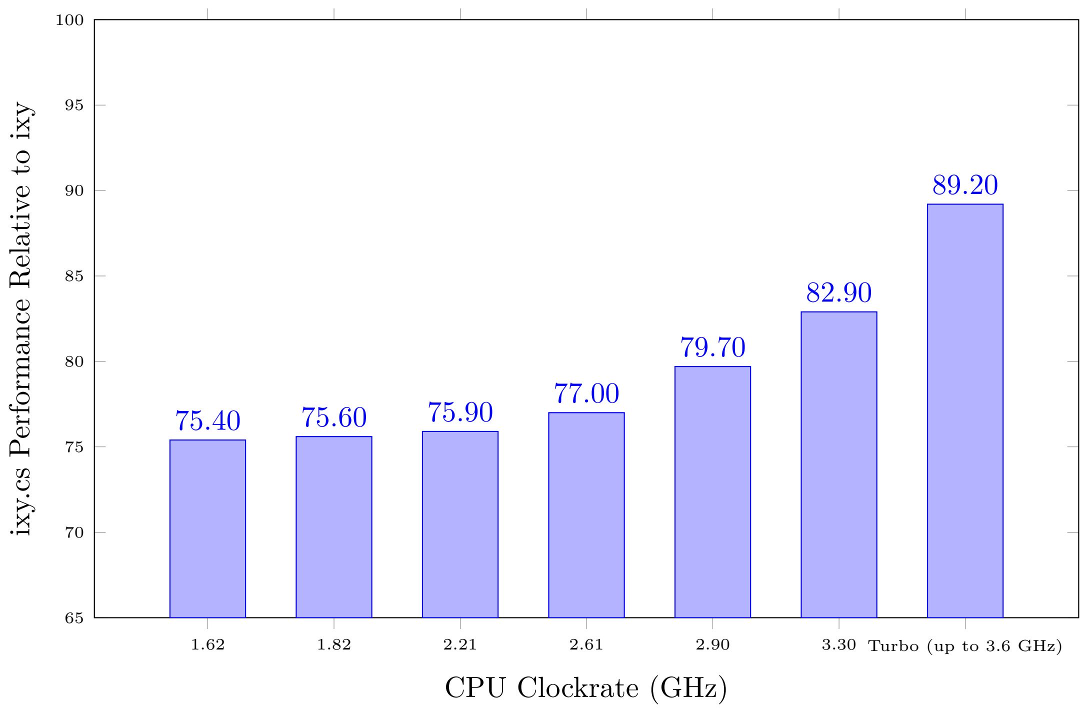

FIGURE 5.2: ixy.cs performance relative to ixy with varying CPU clockrates and a constant batch size of 32

5.2 C# SPECIFIC PERFORMANCE CONSIDERATIONS

5.2.1 PACKET BUFFER ALLOCATION

The first functioning version of ixy.cs provided a packet generator demo before a forwarder was added. This packet generator only sent packets at half the line rate, about 7 Million packets. Obviously this result was unsatisfactory, as the C version easily sends packets at line rate.

The packet buffer demo is very simple and its code can be found in `Demo/PacketGenerator.cs`. At startup packets are prepared and filled with some payload before being returned to the mempool, so they can be retrieved and sent later. In the generator loop, these packets are retrieved from the mempool and then sent by the `TxBatch` function in `IxgbeDevice.cs`, where they are eventually returned to the mempool buffer so the cycle can restart.

The first noticeable problem was that the retrieval of the packet buffers actually took 1.5 times as long as the batching function. This pointed to the most common performance problem that C# applications encounter: excessive allocation and disposal of objects.

Every time the generator retrieved the buffers with the `Mempool.GetPacketBuffers` function, all the required buffers had to be instantiated before being returned to the caller. When buffers were returned to the mempool with `Mempool.FreeBuffer`, the buffer objects were not actually kept in the mempool, but instead only a counter was updated with the new number of free buffers.

This behavior is fine in the C driver, where the packet buffers are allocated in DMA memory and the mempool simply returns pointers to a base address with a certain offset for the correct buffer. However, in C#, where packet buffers are implemented through the pointer objects explained in section 4.4, constantly instantiating new instances of this class is very costly. All heap allocated non-primitive objects that go out of scope in C# eventually have to be reclaimed by the garbage collector. If an excessive amount of objects is created, the garbage collector wastes many CPU cycles trying to keep up with the process' instantiations, which is precisely what led to the buffer retrieval being the costliest part of the packet generator.

The solution to this problem was keeping a stack of packet buffer objects in the mempool. Packets are then retrieved from this stack and returned to it when `FreeBuffer` is called. While new `PacketBuffer` objects still have to be allocated in `TxBatch` and `RxBatch`, this did drastically reduce the required number of allocations.

Another simple but very effective optimization was turning `PacketBuffer` from a class into a struct. In C# there are some differences between a class and a struct. The most relevant difference for ixy.cs is that structs tend to provide better cache locality and are cheaper to allocate. Having an array of `PacketBuffers` classes, each of which only contains a single 64 byte integer representing the address of the DMA memory, only ensures a coherent list of pointers to `PacketBuffer` objects. The integers contained in the objects are not necessarily coherent in memory. An array of structs, however, functions in basically the same way as it would in C. Thus, having `PacketBuffer` as a struct instead of a class boosts performance by improving cache locality and reducing allocation cost. [2] [6]

After these optimizations, performance already improved significantly and packet buffer allocation stopped being a major performance bottleneck.

5.2.2 Accessing unmanaged memory

As explained in section 4.4, the `PacketBuffer` and descriptor objects in ixy.cs are implemented as wrappers around a pointer to the actual buffer or descriptor in unmanaged DMA memory. When the driver or an application writes or reads data from these objects, the write or read operations are executed on this pointer with an offset corresponding to the required field of the buffer or descriptor. The way that these reads or writes occur is another opportunity for optimization.

Initially, I used the safe C# way of writing or reading data to and from unmanaged memory using the Marshal class from the .NET assembly `System.Runtime.InteropServices`. This class has methods such as `ReadInt64, WriteInt64, ReadByte, WriteByte` and so on for various primitive data types. These functions take an `IntPtr` object, which is the safe C# pointer struct, and an offset and read or write the data from or to the resulting address. Using this mechanism, the entire ixy.cs driver was implemented without the use of the unsafe keyword or the unsafe compiler flag, using nothing but managed, safe C# code aside from one C function. The drawback to this method is that the performance is vastly inferior to using raw pointers in C#. C# supports pointers for primitive types, which can be moved and dereferenced, just as in C. In a separate synthetic benchmark, I found that `Marshal.ReadInt32` takes 1.58 times and `Marshal.WriteInt32` 1.73 times as much time as their respective raw pointer equivalents.

Since the performance impact is quite significant, I decided to change the packet buffer and descriptor classes to use raw pointers instead of the `Marshal` class. Classes and structs containing members which are pointers can only be used in unsafe contexts so in order to keep as much safe code as possible, all three of the classes keep an unsigned 64 bit integer representing a pointer, instead of an actual void or byte pointer type. This integer is then cast as a pointer in the properties for accessing various buffer or descriptor fields. Making this change increased the throughput of the forwarder, which had been implemented by now, by almost 50 percent.

While it is unfortunate that using the `Marshal` class is not a viable way to access unmanaged memory, it should be noted that the two ways are actually effectively almost identical. Despite the fact that Marshal.Read or Marshal.Write do not require an unsafe context, they do not provide any guarantees that, for instance, the address that ReadInt32 reads from actually contains an Int32. However, the `Marshal` methods are certainly more idiomatic C# and are easier to read, write and manage than handling pointers directly. The final version of ixy.cs uses the unsafe keyword only three times - exclusively for these packet buffer properties. The ixy.cs API does not expose any pointers to other applications.

The reasons for the overhead of the `Marshal` methods are twofold. First, the functions have some security attributes, which create slightly more work for the runtime. Secondly, the `Marshal` methods support reading and writing aligned and unaligned memory, which means that the functions first have to check the given address to determine whether the data is aligned or not. Unaligned data has to be copied entirely, but even for aligned data, this check introduces some overhead, which is very noticeable for performance critical software like ixy.cs.

Using a third way of accessing unmanaged memory, the `UnmanagedMemoryStream` class, was also evaluated, but found to be entirely inefficient for the driver's purposes. This class was designed for copying sequential data from unmanaged to managed memory and performs unacceptably poorly when used to access random memory in a packet.

Unfortunately, not all of the cost can be avoided when using raw pointers. Accessing unmanaged memory is still significantly more expensive than managed memory because any access to memory outside the runtime's managed domain introduces an overhead and additional work for the runtime. Due to this cost, the driver has to be very conscious of any unmanaged access operations and tries to avoid them more than the C version of ixy does. The increased cost of accessing unmanaged memory is mostly responsible for ixy.cs benefiting from a higher batch size. The driver makes the assumption that each packet in a batch will belong to the same mempool, so the mempool is only retrieved once per batch function. As the mempool ID has to be retrieved from the packet buffer in DMA memory, this retrieval is relatively expensive. Touching the buffers after they are received, in order to simulate a realistic workload, is also significantly more expensive for ixy.cs than it is for ixy, as mentioned in section 5.1. The reason for this also lies in the increased cost of accessing or modifying the packet buffer data.

5.2.3 Further considerations

Another improvement was the use of simple arrays instead of lists or other advanced data structures, where possible, in order to reduce the need for expensive resize or copy operations. Casting in C# is also not without its cost. Even casting primitive types shows a small, but measurable impact. Being conscious of casting and trying to be consistent in the choice of integer types slightly improved performance and throughput.

It is interesting to note that the compiler configuration has an astonishing impact on performance in .NET Core. Compiling the driver with the Debug, as opposed to the Release configuration decreases the throughput by up to 90%. While it would be very interesting to know which specific compiler optimizations cause this difference, unfortunately the .NET Core compiler documentation does not provide any information on the actual optimization steps it performs in the release configuration. The various options for the compiler are in general very sparse and poorly documented, which prevented me from experimenting with different compiler configurations.

As the benchmarks in figure 5.2 show, ixy.cs benefits more than ixy from a higher CPU frequency. The difference in performance between the drivers is increased when the clock rate is reduced and decreases when the Turbo Boost feature is enabled. The reason for this is that the overhead produced by allocation and garbage collection becomes smaller with a faster clock rate. With faster processors this overhead should become less noticeable, moving the performance of ixy.cs even closer to its C equivalent.

Chapter 6

Evaluation And Retrospective

This section aims to answer the question of whether C# is a suitable programming language for driver development, how simple and comfortable the development experience was and whether more drivers should be written in the C# programming language.

6.1 Performance

The first and in many cases the most important consideration is whether C# provides the required efficiency to write drivers without having to suffer from performance drawbacks. While not all drivers have very strict performance requirements, some, such as network drivers or graphics drivers, certainly do. A programming language which is not able to produce code that performs as well as C code, or at least acceptably close to as well, cannot be considered for a production-ready driver for performance critical hardware.

As the results of the performance analysis showed, C# is adequately fast to be used for real world driver development. Depending on the exact benchmark setup, ixy.cs performed relatively close to or in some cases even as well as the C implementation. Whether the measured difference in performance in some setups can be deemed acceptable or not is hard to answer objectively, but the results are certainly good enough to at least make C# worth considering when writing a new driver. This alone is already a significant fact, when considering that virtually every driver for Linux is currently written in C without any other language even being considered. Even user space drivers, which do have the option to choose any programming language still default to using C as if there were no other options to choose from. This evaluation has shown that this stance should be reevealuated and that differences in performance in most cases not large enough to disqualify C# from being the programming language of choice for driver development. For extremely performance-critical scenarios, where the only relevant factor is how efficient the produced code is, C will likely continue to be the option that provides the best results. Despite managed languages getting more and more efficient and runtimes reducing their overhead, it is very rare for code written in these languages to outperform equally optimized C code. However, for many situations where a slight decrease in performance is acceptable, the benefits of writing drivers in C# instead of C, some of which will be explained or reiterated in the next section, may outweigh the relatively minor decrease in efficiency.

That being said, the benchmarks in section 5 have shown that there are certainly also scenarios where ixy.cs performs just as well as the C version. In particular in setups where higher batch sizes are acceptable and a modern CPU with a high clockrate is used, ixy.cs could be used without any significant performance impact compared to ixy. Furthermore, performance concerns are much less relevant in other types of device drivers, such as drivers for peripherals or other devices which do not necessarily have performance as their primary goal. However, these types of drivers do benefit from the additional stability that a type- and memory safe, managed language such as C# can provide.

The performance of the ixy.cs driver could certainly still be improved, as well. As mentioned before, the purpose of this project was to build a simple, educational network driver written in idiomatic and clean C# code, using as few C functions as possible. If its purpose was instead to create the fastest possible user space network driver possible, written mostly in C#, the resulting performance would likely be improved. For instance, by avoiding the `PacketBuffer` class or handling any access to unmanaged memory in C most of the driver's current performance bottlenecks could be largely mitigated. A mixed-language driver could leverage the slightly superior speed of C with the easier development process and improved stability of C# by using C functions for some of the more expensive operations while providing a modern C# interface and most of the benefits that a full C# driver has.

6.2 Development Process

One of the ideas which motivated this thesis was that driver development could potentially be simplified by choosing a more modern programming language than C, which is more simple and comfortable to develop in. As discussed earlier, C# is generally considered to be more simple to work with than C, so it seemed an ideal choice of language for this project. This section discusses whether the development process was in fact less complicated in C# than it would have been in C and how the process differed from more usual C# development.

The size of the drivers' codebases is roughly the equal. Excluding the demo applications, C header files and NIC constant declarations, ixy contains slightly under 1000 and ixy.cs 1180 lines of code. The difference is largely due to the method based descriptors and the wrapper objects around packet buffers that ixy.cs uses.

As discussed in section 2.1, C# provides many convenient features and various syntactic sugar, which is meant to make development with C# easier and safer. ixy.cs uses some of these features to make the code more readable, more maintainable and in some places more compact. The following snippets show two functions from the ixy driver and their C# equivalents.

`ixy_tx_batch_busy_wait from device.h`

```c
static void ixy_tx_batch_busy_wait(struct ixy_device* dev,
  struct pkt_buf* bufs[], uint32_t num_bufs) {
    uint32_t num_sent = 0;
    while ((num_sent += ixy_tx_batch(dev,
      bufs + num_sent, num_bufs - num_sent)) != num_bufs) {
    }
}
```

`TxBatchBusyWait from IxyDevice.cs`

```csharp
public void TxBatchBusyWait(PacketBuffer[] buffers)
{
    int numSent = 0;
    while(numSent < buffers.Length)
    {
        numSent += TxBatch(buffers.Skip(numSent).Take(buffers.Length - numSent)
          .ToArray());
    }
}
```

`wait_clear_reg32 from device.h`

```c
static inline void wait_clear_reg32(const uint8_t* addr,
  int reg, uint32_t mask) {
    __asm__ volatile ("" : : : "memory");
    uint32_t cur = 0;
    while (cur = *((volatile uint32_t*) (addr + reg)),
      (cur & mask) != 0) {
        usleep(10000);
        __asm__ volatile ("" : : : "memory");
    }
}
```

`WaitClearReg from IxyDevice.cs`

```
protected void WaitClearReg(uint offset, uint mask)
{
    uint current = PciMemMapAccess.ReadUInt32(offset);
    while((current & mask) != 0)
    {
        Thread.Sleep(10);
        current = PciMemMapAccess.ReadUInt32(offset);
    }
}
```

(The snippets have been slightly edited to improve formatting. The original versions can be found in the ixy and ixy.cs repositories)

Although readability is very difficult to compare objectively, the C# version of these snippets would likely be considered more expressive by the vast majority of programmers. In particularly the LINQ syntax of `buffers.Skip(numSent).Take(buffers.Length - numSent)` is based on natural language and might in fact even be understood by someone without any programming knowledge. The concept of pointers is not trivial and their use, as in `wait_clear_reg32` can make code difficult to read and maintain. This can lead to mistakes by programmers or maintainers later on. For instance, in line 5 of the third snippet, a simple mistake, such as casting `addr` to a pointer before adding `reg` would cause the address to be wrong and the wrong register being accessed - a bug which could be very difficult to find and might even make its way into a release. The C# code is more clear and makes it easier to avoid or find these mistakes.

While it may seem far fetched to assume that cleaner syntax directly reduces the number of bugs and errors, in large projects, which may contain thousands, tens of thousands or even more lines of code, having this increased clarity is very likely to avoid misunderstandings between developers and make code more maintainable, leading to more stable, less error-prone and safer software.

Not only the internals of the driver, but also its interface and its expandability can benefit from being written in a modern, object-oriented programming language. As mentioned in section 4.1, extending ixy.cs to work with another driver implementation such as VirtIO is easier than extending ixy, due to the fact that ixy.cs can use actual object-oriented inheritance. Having driver implementations inherit from `IxyDevice` is simpler and cleaner than including a reference to an ixy device in each implementation, which will then have to be passed into most driver functions. Something as cryptic as ixy's `container_of` function, which is used as a way to essentially implement inheritance in C through macros, is also not necessary in C# or other object-oriented programming languages.

If a framework built on top of ixy.cs wanted to have, for instance, a UdpPacket class, which represents a UDP packet, this class could simply inherit from `PacketBuffer` and provide additional accessors for UDP-related fields. A TcpPacket could do the same. In functions, which are protocol-agnostic, objects of both types could be processed as `PacketBuffer` objects using polymorphism. This is just one of many examples that show the superior interface that ixy.cs can provide.

It should be noted, that due to its simplicity and scope, there are a lot of powerful C# features that the ixy.cs driver does not actually use. Features like lambda expressions, asynchronicity, generics, interfaces and locks have not been used in the driver but could easily be used in larger projects where the differences between C and C# code become more noticable. In particular C#'s async keyword, which could make the inherently asynchronous `TxBatch` function more usable in an asynchronous context or various .NET concurrency features, which facilitate thread-safety, would be very useful in more advanced versions of the driver. Countless classes and data structures, which the .NET framework provides would also shorten development times compared to C development, where software much more frequently has to implement their own data structures and algorithms.

6.3 C# patterns to be avoided

Some typical C# patterns which are very convenient in other applications should, however, be avoided in driver development for performance-critical systems.

Section 4.4 has discussed the move away from the idiomatic C# way of reading and writing unmanaged code with the `Marshal` class. Unfortunately, this is only one of several occurrences where C# functionality could not be used for performance reasons. Another concession is the relative sparseness of classes or structs in the driver. As discussed above, structs for the RX and TX descriptors were cut from the main branch of the driver in order to reduce the amount of allocations. While this driver's simplicity means that not many classes would be used anyway, the fact that frequent object instantiation is such a performance concern means that larger projects would likely have to be very careful about over instantiating classes and structs. As a result, some parts of the code of very performance critical drivers might look less object oriented than C# code should and instead make use of more C style function-based programming. ixy.cs minimizes the use of "C-like" code as much as possible, however.

Care also has to be taken into using advanced data structures provided by .NET. For instance, `RxBatch` initially used a dynamically sized `System.Collections.Generic.List` object in order to keep buffers it received. At the end of the function, `RxBatch` turned the list into a simple array object with a fixed size. This was an intuitive way of handling the fact that `RxBatch` may not always receive a full batch of buffers. However, turning the list into an array in every call had a negative impact on performance, as the underlying array had to be copied into a new array and the list had to be reclaimed by the garbage collector. Similarly, using a `System.Collections.Generic.Stack` proved to be less efficient than a custom fixed-size stack because it introduced some overhead and performed unnecessary bounds checks on insertion and retrieval of objects.

6.4 Suitability for low-level tasks

The biggest challenge in developing a network driver in C# was the low-level data management. It is very telling that the only part where the driver relies on a C function regards memory allocation and that most of the larger architectural changes that ixy.cs makes are related to memory management. As explained in section 4.3, C# lacks the fine-grained control to allocate memory suitable for direct memory access by the NIC. While it was very simple to write this one function in C and call it from C#, this does show that C# is not quite designed for low-level systems programming. This is not necessarily a bad thing and does not prevent the language from being used for driver development, as this thesis shows. Many technologies and programming languages evolved to be used in areas they were not originally designed for (for instance,the programming language Lua was originally designed by engineers to simplify a data-entry task).

Compared to other managed languages with similar use-cases as C#, the language does a lot to provide several ways to do low-level tasks. With `Marshal, UnmanagedMemoryStream` and raw pointers, it offers three different ways of accessing unmanaged data, for instance. The .NET framework exposes advanced system interfaces and information such as the page size, which ixy needs to translate addresses. It also provides implementations for many low-level C functions, such as `mmap`. In short, C# is flexible enough to be used in systems programming.

However, when handling the direct access of memory, such as when managing packet buffers, the fact that C# was not designed for low level programming becomes a bigger hinderance. For the C version of ixy, packet buffers pose no challenge whatsoever. A struct is declared with a specific memory layout containing a 64 byte header and a variable sized, 64 byte aligned data segment. The driver can then access the buffers through simple pointers. There are no instantiations and no garbage collector to consider, the buffer header fields can be accessed just like any other struct fields and doing so is not more costly than doing the same for any other struct.

While most of ixy.cs is arguably easier to read and and understand than the original C code, the act of accessing packet buffers or descriptors is more complex and also more costly. Since a user space network driver is largely built around allocating and accessing packet buffers and descriptors in DMA memory, C#'s difficulties with these operations are very unfortunate. Although my idea of wrapper objects for packet buffers is relatively easy to understand and masks the added complexity very well, the driver developer still constantly has to be aware of how the access of the buffers actually works and has to think of various performance considerations.

This aspect of the ixy.cs driver somewhat diminishes the advantages of C# driver development. In the most performance critical parts of the driver, the programmer certainly has to think significantly more about performance than the C programmer would. Some creativity is also involved at times where a workaround such as my packet buffer wrappers are required.

6.5 Conclusion

Re-writing ixy in the C# programming language was not without its challenges, but overall a rather pleasant and straight-forward development experience. A large portion of the C code could be directly translated into C# without breaking any conventions or straying away from idiomatic C#. The resulting code looks cleaner and is in some places significantly easier to read. Using object oriented programming techniques, ixy.cs is much easier to extend than the original C version. The objective of providing a C# port of ixy, which is easy to read and shows the inner workings of a user space packet framework was easily achieved and, provided the reader knows C#, ixy.cs may make driver development look easier and make it more accessible than the C version does.

When it comes to developing a production-ready driver to be used in real world applications, the question of whether C# is the ideal language becomes more difficult to answer. ixy.cs has the same functionality as ixy, has no stability or memory issues and in most setups performs similarly to the C implementation, so C# is shown to be a viable option to be considered for network driver development. Whether the performance tradeoff in some setups is acceptable will vary from project to project. As ixy.cs showed, using C# does not necessarily incur a large performance penalty compared to C, but extreme care has to be put into writing efficient code. Specifically when working on low-level memory management, a C# developer has to put significantly more thought and time into development than a C developer would. It therefore becomes less clear, whether the development process is actually easier for a C# driver and the answer will largely depend on the exact type and scale of the project, as well as how significant ideal performance is.

It is my opinion that C# should be strongly considered in particular when writing a user space driver which is either not extremely performance critical, or when writing a large driver framework, such as DPDK, for instance. In less performance-intensive drivers, developers won't have to spend as much time optimizing C#-specific performance bottlenecks and the downsides of using C# will become less significant. In large scale projects, a driver could easily be written largely in C#, utilizing more of the language's and .NET features than ixy.cs does, while keeping the very performance-critical memory management part of the driver in C. In a multi-language project some care will have to be put into finding an efficient way of coupling C and C# seamlessly without too much accessing of unmanaged memory, but building a high performance driver will be easier than in pure C#. The driver can then still utilize all the benefits of C# that have been mentioned in this thesis, including cleaner syntax, object oriented techniques and a vast amount of functionality from the .NET framework. In particular a large scale driver with some layer of abstraction, designed for the .NET ecosystem could benefit greatly from utilizing C# instead of relying entirely on C.

In conclusion, C# has been shown to be a viable alternative in user space driver development. Although the language brings some additional challenges, it excels at delivering clean, readable and maintainable code, which can easily be extended and built upon, while providing competitive performance. Considering C# for future user space drivers could result in safer and more stable systems and at least partially eliminate the need for inherently less safe C code.

Chapter A

List of acronyms

DMA	Direct Memory Access.
DPDK	Data Plane Development Kit.
NIC	Network Interface Controller.
RX	Receiving.
TX	Transmitting.

Bibliography

[1] Paul Emmerich et al. “User Space Network Drivers”. [accessed on 08/01/2018]. URL: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ixy_paper_draft2.pdf.

[2] “C# Language Specification (Standard ECMA-334)”. [In particular section 8.8 - Accessed on 08/01/2018]. 2001. URL: https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-334%201st%20edition%20December%202001.pdf.

[3] Intel Networking Division. “Intel 82599 10 GbE Controller Datasheet”. [Accessed on 08/01/2018]. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[4] Galen Hunt et al. *An Overview of the Singularity Project.* Tech. rep. 2005, p. 44. URL: https://www.microsoft.com/en-us/research/publication/an-overview-of-the-singularity-project/.

[5] Stack Overflow. “Stack Overflow Developer Survey 2018”. [Accessed on 08/01/2018]. 2018. URL: https://insights.stackoverflow.com/survey/2018/.

[6] Emmanuel Schanzer. “Performance Tips and Tricks in .NET Applications”. [Accessed on 01/09/2018]. 2001. URL: https://msdn.microsoft.com/en-us/library/ms973839.aspx.

[7] Intel DPDK Validation Team. *DPDK Intel NIC Performance Report Release 18.02.* Tech. rep. [accessed on 08/01/2018]. 2018. URL: https://fast.dpdk.org/doc/perf/DPDK_18_02_Intel_NIC_performance_report.pdf.