Department of Informatics
Technical University of Munich

# TECHNICAL UNIVERSITY OF MUNICH

## DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

# Writing Network Drivers in Go

Sebastian Peter Johann Voit

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

# Writing Network Drivers in Go

# Netzwerktreiber in Go schreiben

| | |
|---|---|
| Author: | Sebastian Peter Johann Voit |
| Supervisor: | Prof. Dr.-Ing. Georg Carle |
| Advisor: | Paul Emmerich |
| Date: | October 15, 2018 |

## Abstract

Network drivers, as many other drivers, are the link between hardware and software. Drivers are operating system specific software and function as an interface in between a physical device and software that use the device. Unfortunately any actual knowledge about them is rare, even amongst programmers that use them daily in production environments. As an understanding of the full network stack is important for optimizations, performance, debugging, etc. , it is essential that information about drivers is readily accessible in order to foster networking as a science, a profession and a hobby.

Network Interface Controller (NIC) drivers used to be pieces of software that was locked away deep within the kernel and was rarely touched, but even with the rise of user space drivers such as DPDK or Snabb, it is still mostly treated as a black box that increases speed. Knowledge cannot be forced onto people, but access and presentation can be improved in order to lower the entry barrier. ixy, a simple user space driver for ixgbe devices, was created as an easy to understand example that presents the basic functionality of such drivers in a clearly laid out manner.

C was chosen as the lowest common denominator, a language that every programmer should have at least come into contact with on a basic level. While C has many properties that make it a language well suited for drivers, some of its flaws lie in the dangers of programming errors and prevention measures can complicate the code in a way that it is not easily understandable for programmers not used to it. In order to increase the variety in languages and compare their efficiency, we present a Go implementation of the ixy network driver.

This Go implementation, called ixy.go, ties to reimplement the original driver as closely as possible in idiomatic Go and does so within ≈1000 lines of code. Compared to C, Go offers improved security and easier to read code as well as multiple other features that make the effort of the implementation worthwile. With 24.9803 million packets per second it is ≈10% slower then the C implementation while offering various advantages in other areas as a high-level language. The advantages and disadvantages of Go as a network driver language will be discussed and evaluated whether it is a viable consideration.

# Contents

# Chapter 1

# Introduction

## 1.1 Kernel modules vs User space drivers

Generally, a driver can be written in one of two ways: as a kernel module or as a user space driver. Both have distinct advantages and disadvantages though ultimately user space drivers are faster, which is the most important requirement in modern use cases.

A kernel module runs in the kernel space. This entrusts it with certain privileges, most notably hardware interrupts and protection from user influences. Running in the kernel space does however come with some serious downsides. Kernel calls require a computational overhead due to context switches, therefore slowing down the software considerably, and has generally proven to be simply too slow for specialized applications. Additionally kernel development is a lot more complex and labour intensive as many of the tools used for debugging such as gdb and Valgrind are not available and it generally being more cumbersome. Also when writing kernel modules, the choice of programming language is restricted to a subset of C. C programs tend have security issues that arise from poorly written and tested code. With the absence of many debugging tools one has to be extremely careful, as errors in the kernel space can have much more severe consequences then errors in the user space. Where a user space driver would produce a segmentation fault, a kernel module could overwrite arbitrary memory locations, potentially corrupting other data, including the kernel itself.

User space drivers on the other hand mitigate or straight up do not have these kind of problems and solutions tend to be simpler and carry less risks due to having less privileges. User space drivers can be written in any language of choice as long as a compiler exists and programmers can make full use of all tools created for the language

which helps with debugging and other tasks such as performance evaluation. Coupled with inherent safety measures of the chosen language, this results in generally safer software. Additionally this approach grants the application full control over the driver which can lead to a more optimized integration of an application into the driver and hardware.

Still, these are not the main selling point of user space drivers. While certainly helpful, speed is the ultimate argument for choosing a user space driver over a kernel module. Kernel code constantly has to switch between user and kernel mode, which poses a significant computational overhead, especially in the case of network drivers due to the frequent copying of data. A user space driver runs entirely in the user space and therefore does not need any context switches or copying which significantly reduces overhead and therefore leaves more time for actual packet handling. The Data Plane Development Kit (DPDK) is a user space driver originally implemented by Intel and later handed over to the Linux foundation. In their packet forwarding performance evaluation Intel claims that on ixgbe NICs, it can operate close to or even at line rate, depending on the packet size [24].

This gain in speed, as demonstrated by user space drivers such as DPDK, is the reason why many packet processing frameworks that used to be kernel-based moved to the user space [8].

## 1.2 ixy.go

ixy is the name of the original C driver, derived from the term ixgbe. As we implement a Go translation of this driver, we call it ixy.go. The speed of user space drivers is an important part in many modern networking applications and therefore are often added to existing projects in order to increase speed. Still most programmers treat it like a black box that somehow increases speed and therefore do not understand the inner workings of such a driver and why they are faster then kernel based drivers.

Since a clear understanding of the full network stack is important when working on networking applications for multiple reasons such as optimization and debugging, ixy has been developed as an educational piece of software that allows interested programmers to understand the basic user space network driver functionality in an uncomplicated manner [8]. Its goal is to make knowledge about such drivers accessible to everyone and therefore was written with a focus on simplicity. ixy shows that it is possible to implement a basic user space network driver in just below 1000 lines of code. While

architecturally based on DPDK and Snabb, it is stripped down to the bare minimum but still optimized on speed where the price of complexity is low.

The C programming language has been chosen as the language for the original driver since it is the language that most programmers have learned at least on a basic level. C is also rather powerful as a language for low level operations though it has proven to be rather difficult to write correctly given the easiness at which a plethora of common programming errors occur, mostly due do obliviousness or forgetfulness. Tools and techniques to combat these errors do exists but usually also make the code seem more complex than it actually is. Therefore it might prove difficult to read and understand for programmers that are neither advanced in C programming nor involved in driver development, making ixy harder to understand then necessary and thus raising the entry barrier.

In order to lower the barrier and make ixy more available to programmers it has and will be rewritten in multiple languages. For this thesis the language of choice was Go. While originally intended to be a networking language for the higher levels of networking with native library support for a wide array of needs, driver level programming has to make due mostly with more basic tools making the implementation an interesting challenge. Go does however offer some notable advantages compared to C. While its syntax is designed very similar to C on purpose, some fundamental concepts were revised specifically for the purpose of readability. It is strongly typed and has a built-in runtime included in the compiled binaries. This runtime adds features such as garbage collection as well as type and memory safety. Despite being oriented towards higher levels, Go does offer the use of lower level constructs like pointers, though no pointer arithmetic, and through the use of syscalls memory management. These properties make for an interesting choice of language for an educational network driver with distinct advantages and disadvantages compared to C.

In the following thesis we analyse the viability of Go as a systems programming, and more specifically driver programming, language and evaluate it and its distinct advantages, challenges and drawbacks compared to other user space drivers and specifically the original C version.

# Chapter 2

# Why Go?

Go, also called Golang, is a statically typed, compiled and imperative programming language developed by Google. It has intentionally been developed to have a syntax similar to the C programming language with input and ideas from other languages [18]. Go compiles to statically linked binaries with very fast compile times, making it independent of libraries after compilation. Compilers for Windows, macOS and Linux distributions exist. As for the Go implementation of ixy we tested as far back as version 1.10.2 and due to the backwards compatibility promise [17] our driver will most likely be compatible with all versions of Go 1 thereafter.

The Go binaries always include the Go runtime. It implements "garbage collection, concurrency, stack management, and other critical features of the Go language" [18] and therefore provides central functionality to the programs, though it is not to be confused with the virtual machine approach of the Java runtime. This runtime adds safety features as an improvement to combat many of the common programming errors such as memory safety for index bounds and pointers. Go offers an extensive native library which provides a multitude of features ranging from so called slices and interfaces to algorithms and other helpers. Though Go is not an object oriented programming language, it does offer the ability to define methods on types, including struct types. Go also offers an interesting approach to concurrency through the use of goroutines, which are based on Communicating Sequential Processes (CSP) ideas, in order to simplify the complexity of other concurrency models. Additionally the main way of exchanging information are channels instead of sharing memory in order to further simplify concurrency. This approach is embodied in the slogan "Do not communicate by sharing memory; instead, share memory by communicating." [16] It is noteworthy that many of the security features the runtime enforces can be bypassed, mainly through the use of

the unsafe package. This package implements arbitrary pointers that allows programmers to defeat the type system and read from and write to arbitrary memory. Since the runtime is a central element of Go and it is ixys' goal to be an idiomatic program, we try to minimize the use of this package as much as possible unless the idiomatic solution to a problem makes use of it. And example for that would be determining the endianess of a machine.

As already mentioned, the syntax of Go is intentionally similar to C. However some notable changes have been made in order to improve readability. Two changes become apparent immediately upon first seeing a Go program:

1. Go does not have semicolons at the end of a line. Since trailing semicolons don't serve any real purpose except denominating the end of a line, removing is from the language was a central syntax design goal in order to prevent unnecessary mistakes in that regard. More specifically semicolons are still used as terminators in other places such as separating arguments of a for loop. Technically they are also still needed at the end of lines but may be, and virtually always will be, omitted and will then be inserted automatically during compilation [22].

2. Variable names before type. The approach of the C programming language was unusual at the time and had one distinct advantage: a declaration is an expression and one can state the type of the expression. The downside however becomes apparent when a statement becomes more complex, as it can become hard to read, in fact C expressions are read in a spiral. Go reverses this order therefore making a simple declaration readable from left to right. This extends to statements of any length and complexity, one can read them from left to right without needing to stop thus avoiding the C gibberish issue altogether.

Other minor differences such as type inference exist as well but the general syntax and structure is very similar compared to C code, Go just tends to be easier to read. Additionally many potentially confusing C constructs can be replaced by easier to understand Go features.

Most languages don't have a fixed style but instead multiple third party style guides such as the Linux kernel coding style [23]. Combined with the unique preferences of each programmer this results in a plethora of different styles. Due to the aforementioned automatic insertion of semicolons, a minimal amount of fixed formatting in Go is unavoidable. But even beyond that, Go strives towards a unified formatting style as this is one of the "most [contended] but least consequential [issues]" [16]. In order to reach the goal of a unified coding style without sparking debates, the approach is in the form of a tool that takes care of formatting instead of a "long prescriptive style

guide" [16]. The "gofmt" program takes files and packages and ensures that the output conforms to the Go coding style.

"gofmt" is just one of multiple tools that come with the Go programming language. The standard Go installation includes a variety of other tools, ranging from profiling to disassemblies of executables to automatic documentation to the support of C code within Go packages and vice versa. This array of supportive tools aid in many aspects of programming, making the development process easier, faster and safer.

Go is not without flaws though and they have to be accepted when using it as the language of choice. The main point concerning ixy is that the garbage collection overhead can severely limit the viability in systems programming and since there is no manual allocation and freeing of memory it is not optional. While this area has been the target of many improvements over the last years, it still does not satisfy hard real time requirements [10]. Go also does not have the keyword volatile or, at least to our knowledge, an equivalent that prevents optimization concerning a variable or special memory. There are other disadvantages, such as the lack of generics, though most of them are hardly relevant to ixy.

# Chapter 3

# Related Work

In the previous Chapter we discussed what factors can be detrimental to Go being a suitable languages for network drivers as well as the advantages it offers as a high-level language. While Go is used for a great number of networking operations, we failed to find any example of it being used for user space driver programming.

Not even Googles new operating system Fuchsia makes use of Go for their drivers. It does however use Netstack [12], a userland network stack written in Go as part of its garnet layer [14]. The Garnet layer is on top of Zircon [15], a microkernel and the foundation upon which Fuchsia is build [2]. Zircon includes all the ethernet drivers which are exclusively implemented in C and C++[1]. ixy.go is therefore the first network driver written in Go.

Biscuit [3] is an operating system kernel written in Go in order to evaluate the impact that writing a POSIX kernel in a high-level language has compared to C. The premise of this paper is similar to our thesis but has a different subject. A monolithic kernel also needs network drivers and driver support is implemented for Intel PCI-Express Ethernet NICs, which are the same NICs we support: ixgbe. The main difference is that Biscuit as an operating system implements a kernel level driver while we implement a user space driver. Unfortunately the paper simply mentions the networking components implemented with no further analysis as its focus is on the scope of the full kernel. Conclusions for Biscuit are that a Go kernel is ≈15% slower but is at many points easier to implement and has increased security as type- and memory safety prevent

[1] `https://fuchsia.googlesource.com/zircon/+/master/system/dev/ethernet/`

real-world bugs. These features make Go a viable consideration when speed is not the top priority.

# Chapter 4

# Driver Architecture

Since ixy.go is a port of the original ixy driver it tries to stick with its architecture rather closely. We will often refer to the original driver [5] and the corresponding paper [8] when discussing architecture and implementation in the following. All necessary information will be covered briefly but we focus on the ixy.go specific implementation. We therefore recommend the ixy paper in case more extensive information is desired and will refer to it in the following sections. Our ixy.go implementation is available under the following GitHub page: `https://github.com/ixy-languages/ixy.go`. Unless specified otherwise, any code that is mentioned refers to commit `4145aa8` in the `master` branch of this repository. Instructions for installation and usage can be found in the `README.md` file.

The Go port of the ixy driver is in most parts architecturally identical to the original driver which in turn is based on the official Intel ixgbe driver and based on ideas from both DPDK [13] and Snabb [1]. Snabb inspired the initialization as well as the operation without loading a driver and the Application Programming Interface (API) based on explicit memory management, batching and abstraction was modeled after DPDK.

The Intel 82599 datasheet [11] explains the process for initialization of the driver, how the receive and transmit queues are set up and operated and thus was essential in development. All page and section numbers refer to revision 3.3 (March 2016) of the 82500ES datasheet. The extensiveness of the datasheet was one of the main reasons ixgbe was chosen. We will refer to the datasheet when discussing the implementation. The primary design goal was simplicity as ixy is supposed to be an educational piece of software. This means that compared to other userspace drivers it was trimmed down to the bare essentials and therefore missing features such as various means of offloading,

making it at least one order of magnitude simpler. The driver was then optimized for speed, as long as the cost in complexity was not too high, meaning that the driver is not as highly optimized as other user space drivers. The driver is not meant for production use whatsoever, it is meant for education only. The API was also not designed in the name of user friendliness but rather aims to make the inner workings of the driver clear to application programmers. A higher level of abstraction could be built onto the current API and functionally could also be added but would go against the principle of education and simplicity and thus was cut from the ixy drivers.

As mentioned before, we will focus on the differences between the original driver written in C and our rewrite in Go. As additional ports in different languages have already been done and more are in the works, these thesis will provide a comparable baseline of the performance of various languages in the area of systems programming and more specifically network driver programming.

## 4.1 Overview

From an outside perspective the driver works as follows: An application first needs to initialize a number of devices to operate on which it can then use for sending and receiving packets. All packet manipulation has to be done by the application though the received or self-initialized packet buffers.

## 4.2 Driver Abstraction

All versions of ixy feature only one layer of abstraction: Decoupling of the driver and the user application. Applications call into ixy for initialization and ixy chooses the appropriate driver automatically and returns it. The original C implementation has an `ixy_device` struct (in `device.h`) for this purpose that includes information common amongst all driver implementations as well as function pointers to functions the API then has to provide. The returned struct will then be of the type of the driver for the NIC, an `ixgbe_device` (in `ixgbe.h`) for ixgbe and a `virtio_device` (in `virtio.h`) for the VirtIO, the two currently implemented drivers in the C version. These structs include the `ixy_device` alongside the driver specific information and functions.

This setup certainly works but can be done more elegantly and readable in Go. While this would be a prime case of inheritance in object oriented languages, Go is not an object oriented language. It does however offer receiver functions and interfaces which

can be used in order to achieve a solution similar to such languages. Receiver functions are functions defined on a type or pointer type while an interface specifies a number of method signatures and is simultaneously a type itself. Every type that satisfies all of the method signatures of an interface implements this and possibly other interfaces and thus can be used whenever the implemented interface type is required. This is also what the infamous and often misunderstood empty interface provides: since every type implements at least zero methods, every type implemets an empty interface.

In our case though this concept provides us with a convenient way of modelling different drivers: as in C we use an `IxyDevice` struct (in `device.go`) that will be part of the driver specific structs but instead of function pointers we define an `IxyInterface` in `device.go` with method signatures that all drivers must implement.

```
1 type IxyInterface interface {
2         RxBatch(uint16, []*PktBuf) uint32
3         TxBatch(uint16, []*PktBuf) uint32
4         ReadStats(*DeviceStats)
5         setPromisc(bool)
6         getLinkSpeed() uint32
7         getIxyDev() IxyDevice
8 }
```

Functions and variables starting with a capital letter are exported in Go which we make use of to further control our API and only export necessary functions of the interface. Depending on the device at the given Peripheral Component Interconnect (PCI) address, the corresponding driver initialization is then invoked which, as in the C driver, will return the driver struct which is then handed back as an IxyInterface. This results in much more readable code and better extensibility as new drivers can be added simply by inserting the file into the Go package and adding a condition to the `IxyInit()` function in `device.go` under which the new driver should be invoked, no further changes needed.

## 4.3 Initialization Process

Next we will take a look at the initialization process. Since this works mostly similar to the C implementation, we will give a short rundown of the process and only get into greater detail when we need a different approach in Go. The initialization process starts with a call of `IxyInit()` which then in turn calls the initialization function of the driver that corresponds to the PCI device at the address that has been handed over. In the case of this thesis we only support ixgbe devices and no VirtIO. `ixgbeInit()` in `ixgbe.go` then starts by initializing all fields of the struct and then calls `resetAndInit()` in `ixgbe.go`. This part of the NIC initialization is described in Sections 4.6.3 and following

of the datasheet. After setting the correct values for the registers, which are explained in Section 8.2 in the datasheet and are defined as constants and functions in `type.go`, we initialize the link, the status and the receive and transmit queues and enable promiscous mode in order to receive multicasts as well, though we keep interrupts disabled since we rely on polling. Then we wait until the link is online.

```
1  func (dev *ixgbeDevice) resetAndInit() {
2          setCReg32(dev.addr, IXGBE_EIMC, 0x7FFFFFFF)
3          setCReg32(dev.addr, IXGBE_CTRL, IXGBE_CTRL_RST_MASK)
4          waitClearCReg32(dev.addr, IXGBE_CTRL, IXGBE_CTRL_RST_MASK)
5          time.Sleep(time.Millisecond)
6          setCReg32(dev.addr, IXGBE_EIMC, 0x7FFFFFFF)
7          waitSetCReg32(dev.addr, IXGBE_EEC, IXGBE_EEC_ARD)
8          waitSetCReg32(dev.addr, IXGBE_RDRXCTL, IXGBE_RDRXCTL_DMAIDONE)
9          dev.initLink()
10         dev.ReadStats(nil)
11         dev.initRx()
12         dev.initTx()
13         for i := uint16(0); i < dev.ixy.NumRxQueues; i++ {
14                 dev.startRxQueue(int(i))
15         }
16         for i := uint16(0); i < dev.ixy.NumTxQueues; i++ {
17                 dev.startTxQueue(int(i))
18         }
19         dev.setPromisc(true)
20         dev.waitForLink()
21 }
```

The initialization is described in Section 4.6.4 of the datasheet, we set the bits in the AUTOC register and then restart the link. The status is initialized by reading from the NIC and discarding the result to clear the fields. Initializing the receive (Section 4.6.7 of the datasheet) and transmit (Section 4.6.8) requires a bit more work:

`initRx()` in `ixgbe.go` operates according to the datasheet but we need additional information for the driver in order to manage the receive queues that hold the packets. For that purpose we allocate DMA memory, which will be explained in the next section, and start to initialize the queue struct. C offers unions that can be used to represent the queue descriptors but since Go forgos unions we have to resort to the use of raw byte slices that we subdivide the dma memory into.

`initTx()` in ixgbe.go works similarly. We follow the instructions of the datasheet and get the data for our driver, which hardly differs from the receive initialization.

After initialization, the queues have to be started. Refer to `startRxQueue()` in `ixgbe.go` for this process. For the receive queue this is done by allocating a mempool that can hold the desired amount of packets and allocate packets within this pool for each queue. For each allocated packet we hand its physical address over to the NIC by using the packet descriptors. Once all the queues are allocated we enable the corresponding queue and tell the NIC that we are done. Starting the transmit queues is easier since they start

out empty, thus the only thing we have to do is to enable them. The implementation is in `startTxQueue()` in `ixgbe.go`.

For the last step we enable promiscous mode which is simply inverting the corresponding flag. We then wait until the link comes back up at which point the driver is good to go.

Communication with the NIC requires reading and setting registers. Ixgbe NICs expose their configuration, statistics and debugging registers via the BAR0 address space. All Base Address Register (BAR) are exposed via the sysfs pseudo file system and can simply be mmap'ed into a privileged processes address space. The register offsets are listed in the datasheet [11], the mapping can be found in `pciMapResource()` in `pci.go` and the getter and setter methods in `regs.go`. Additionally we prepared `read/writeIoXC()` functions in `regs.go` mainly for a ViotIO implementation, the reasoning can be found in the original ixy paper [8], but also find use during initialization.

We originally implemented these functions in pure Go and tried to force reads and writes through empty C calls as Go does not have the volatile key word or an equivalent for our needs as far as we know. This resulted in strange behaviour: while initialization seemed to work, including the corresponding register getters and setters, the program failed later when the NIC never set flags that were necessary to continue. We invested a lot of time in this specific problem but could not find a reason for this behaviour. With the help of cgo, which enables the use of C code in Go programs, we imported the corresponding functions in `device.h` alongside `log.h` of the original C driver and replaced our go functions. This fixed the aforementioned problem though we never were able to provide an exact explanation due to limited time constraints.

```
1  // #include <device.h>
2  import "C"
3  import (
4          "os"
5          "unsafe"
6  )
7  func setCReg32(addr []byte, reg int, value uint32) {
8          C.set_reg32((*C.uint8_t)(unsafe.Pointer(&addr[0])), C.int(reg), C.uint32_t(value))
9  }
10 func readIo32C(fd *os.File, offset uint) uint32 {
11         return uint32(C.read_io32(C.int(int(fd.Fd())), C.size_t(offset)))
12 }
13 func writeIo32C(fd *os.File, value uint32, offset uint) {
14         C.write_io32(C.int(int(fd.Fd())), C.uint32_t(value), C.size_t(offset))
15 }
```

## 4.4 DMA Memory Allocation

We have seen the general process of the initialization, including the receive and transmit queues. This operation requires the allocation of DMA memory: reads and writes within this memory area become reads and writes to the NIC through the use of the mmap function on the desired memory area. In order to use DMA we need to explicitely enable it as it is done in `enableDma()` in `pci.go`.

First of all each queue needs descriptors, in our case we use the advanced receive (data sheet Section 7.1.6) / transmit (Section 7.2.3) descriptors. These descriptors serve as the communication interface between the NIC and the driver, providing information and taking commands concerning packet buffers. Additionally we also need DMA memory for the packet buffers themselves. The `MemoryAllocateMempool()` function in `memory.go` builds upon allocated dma memory and manages this memory as a memory pool that consists of packet buffers.

Now for the actual memory allocation, refer to `memoryAllocateDma()` in `memory.go` for the implementation. As in C, memory mapping a file in Go can be done with one single `Mmap()` method that the syscall package provides [21]. Contrary to C the return value of this function is not a pointer to the start of a memory area but instead a byte slice that contains the memory area.

```
1 err = fd.Truncate(int64(size))
2 //error handling
3 var mmap []byte
4 mmap, err = syscall.Mmap(int(fd.Fd()), 0, int(size), syscall.PROT_READ|syscall.PROT_WRITE,
      ↪ syscall.MAP_SHARED|syscall.MAP_HUGETLB)
5 //error handling
```

Though we could certainly handle this problem similar to the fashion of the C implementation by the extensive use of pointers, this would require the use of unsafe operation in order to circumvent the type system and the runtime. However we do not expect a significant increase in speed through this approach and it would not conform to the goal of writing idiomatic Go. Thus we used the provided byte slices which results in different handling of DMA memory compared to C, hardly more difficult to understand but with different approaches.

In addition to mmapping we also have to take into account that our driver operates on virtual addresses while the NIC needs physical addresses for its operations. We therefore have to not only be able to calculate the mapping between physical and virtual addresses but also have to make sure that this mapping stays consistent thus resident in pyhsical memory. We can use `Mlock()` from the syscall package [21] to disable

swapping and guarantee that the page stays in memory. It does not however guarantee that the physical address stays the same as the page migration mechanism can change the physical address of a memory backed page allocated from the user space at any given time [4]. Fortunately Linux does not implement page migration for explicitly allocated hugepages which we can thus use in order to create a stable mapping. Hugepages are enabled by executing setup-hugetlbfs.sh before starting the driver. This is standard procedure in all user space drivers [8].

Lastly we need to translate virtual into physical addresses. All the necessary information is contained in the `procfs` file `/proc/self/pagemap`, the implementation of this translation is implemented in `virtToPhys()` in `memory.c`. The translated addresses are stored in the queue and can be communicated to the NIC through the usual registers.

## 4.5 Mempools and Packet Buffers

As we have seen in the previous section, a queue has a memory pool in addition to its descriptors. The memory pool is responsible for managing the allocated DMA memory which is divided into fixed size buffers. In the C version a mempool is effectively a stack of packet buffers which are data structures stored in DMA memory that contain a header and the packet itself. The mempool data structure of out Go implementation is of similar fashion and contains metadata for the mempool such as size and the number of entries as well as the slice that points to the DMA memory that is used for packet buffers.

The header of a packet buffer is 64 bytes and contains metadata such as the physical address, the ID of the corresponding mempool and its size. The remainder of the memory area is reserved for the packet data. In Go there is no such a struct for multiple reasons. First of all this approach works in C as, because of its weaker type system, one can simply cast a pointer to DMA memory into a pointer to a packet buffer and thus interpret the underlying bytes accordingly. In Go this would require the usage of unsafe pointers from the unsafe package which we try to avoid wherever we reasonably can. Additionally it would prove difficult to construct a Go struct that accurately represents the desired memory layout as Go does not allow the declaration of an array of unspecified length for the packet data. Slices are also not adequate as a slice is more akin to a pointer to an array with additional information about boundaries and thus cannot be used where the actual data needs to be. We could have devised packet buffers that are allocated outside the DMA memory and contain the metadata as well

as a pointer to the packet data in DMA memory but we decided against decoupling these two parts and spreading them over the memory.

We instead used another approach that manipulates the DMA memory into the same shape as the C implementation does, namely metadata and packet data in DMA memory. We simply address the byte slices at the correct offsets. So instead of pointer casting and addressing the memory as a struct, we set the bytes manually with the help of the binary package [19]. It has to be noted that we do need to know the endianess of the machine for this to work. In this case the idiomatic approach makes use of unsafe pointers, see `lines 34-38` in `device.go`. Unfortunately this approach does not work in Go without an additional data structure as we have to bundle the subslice that represents our packet buffer and a pointer to the mempool together. Simply storing the address of the mempool in the metadata header as bytes, like the C implementation does, causes the garbage collector to target the mempool as a re-translation does not satisfy the criteria to keep the target.

```c
//C implementation: pointer casting
for (uint32_t i = 0; i < num_entries; i++) {
	mempool->free_stack[i] = i;
	struct pkt_buf* buf = (struct pkt_buf*) (((uint8_t*) mempool->base_addr) + i * entry_size)
	    ↪ ;
	buf->buf_addr_phy = virt_to_phys(buf);
	buf->mempool_idx = i;
	buf->mempool = mempool;
	buf->size = 0;
}
```

```go
//Go implementation: byte slice
for i := uint32(0); i < numEntries; i++ {
	mempool.freeStack[i] = i
	pBufStart := i * entrySize
	if isBig {
		binary.BigEndian.PutUint64(mempool.buf[pBufStart:pBufStart+8], uint64(virtToPhys(
		    ↪ uintptr(unsafe.Pointer(&mempool.buf[i*entrySize])))))
		binary.BigEndian.PutUint64(mempool.buf[pBufStart+8:pBufStart+16], uint64(uintptr(
		    ↪ unsafe.Pointer(mempool))))
		binary.BigEndian.PutUint32(mempool.buf[pBufStart+16:pBufStart+20], i)
		binary.BigEndian.PutUint32(mempool.buf[pBufStart+20:pBufStart+24], 0)
	} else {
		//same operations with binary.LittleEndian
	}
}
```

During the initialization exactly one mempool is allocated per queue and pre-filled with empty packets for performance reasons. We then link packet buffers and descriptors by writing the packets' physical address into the descriptors. After enabling the queue through the registers the initialization is complete.

## 4.6 Receiving and Transmitting Packets

Now that we covered the setup of the ixy.go driver, we will take a look at the workload it has to deal with which can be boiled down to two tasks, namely receiving and transmitting packets. For purpose of packet transfer, NICs expose circular buffers called queues or rings. As in the example applications, the simplest setup uses one receive and one transmit queue but multiple queues are also possible. These ring buffers are the DMA descriptors we explained in Section 4.4. Once a ring is initialized, sending and receiving is done by passing ownership of the descriptors between the driver and the hardware. For that purpose the hardware controls a head pointer and the driver a tail pointer which are accessible through device registers. Consult the second passage of Section 4.1.1 of the original ixy paper [8] for further details.

Receiving is implemented in `RxBatch()` in `ixgbe.go` as described in sections 1.8.2 and 7.1 of the datasheet. Our implementation is mostly similar to the original ixy, safe for adjustments to fit the data structures we chose for the Go implementation. After startup the ring buffer is filled with physical pointers to packet buffers as described in the previous section. Whenever a packet is received, the NIC will set the "descriptor done" flag, signalling that it can now be handled by the driver. The driver then reads the whole descriptor, returns the received packet to the application, allocates a new packet from the mempool and resets the descriptor by writing the packets physical address to the descriptor. Once receiving is done, we notify the hardware by moving our tail pointer. Since the NIC works with physical addresses and the driver with virtual ones, we also need a way to translate them. Instead of consulting the pagemap once again, we keep a copy of the ring that is populated with the virtual addresses and use this as our lookup table.

```
//RxBatch(); shortened for the sake of briefness, refer to source code for detailed explanation
for ; bufIndex < numBufs; bufIndex++ {
	descPtr := queue.descriptors[rxIndex].raw
	//status := bits 64 - 95 of the advanced rx recieve descriptor
	if status&IXGBE_RXDADV_STAT_DD != 0 {
		//status IXGBE_RXDADV_STAT_EOP not supported
		desc := make([]byte, len(descPtr))
		copy(desc, descPtr)
		buf := queue.virtualAddresses[rxIndex]
		//copy length from descriptor to packet with encoding/binary
		newBuf := PktBufAlloc(queue.mempool)
		//reset descriptor by writing the physical address of the buffer into the first 64
		    ↪ bit and set the rest to 0, again with encoding/binary
		queue.virtualAddresses[rxIndex] = newBuf
		bufs[bufIndex] = buf
		lastRxIndex = rxIndex
		rxIndex = wrapRing(rxIndex, queue.numEntries)
	} else {
		break
	}
}
if rxIndex != lastRxIndex {
	setCReg32(dev.addr, IXGBE_RDT(int(queueID)), uint32(lastRxIndex))
	queue.rxIndex = rxIndex
}
return bufIndex
```

Receiving as well as transmitting is done in batches, hence the function names. This results in a significant performance increase as otherwise the device register that controls the tail would have to be updated after every packet. Since a write to a register is a comparatively expensive operation, reducing it to once per batch cuts a significant portion of this overhead and thus severely increases throughput. A receive batch always includes as many packets as it has access to while the transmit function operates on a fixed batch size.

The transmit functionality is implemented in `TxBatch()` in `ixgbe.go` as described in Sections 1.8.1 and 7.2 of the datasheet. While packet transmission does follow the same concept and API as packet reception, it is somewhat more complicated due to the asynchronous nature of the NICs' interface, as a packet placed into the ring is not immediately transferred. As blocking would be detrimental to the performance, the receive function is instead divided into two steps.

In the first step we clean up the descriptors that were sent out by freeing the corresponding packets and in the second step the packets are placed into the ring. As with the receive function, this is equally similar to the original C implementation, again safe for adjustments that result from different data structures. The cleaning process starts with calculating the amount of descriptors that can be cleaned, which is based on batch size, and freeing them based on the "descriptor done" flag of the status. The virtual

addresses are once again obtained through a copy. Freeing packets is implemented in `PktBufFree()` in `memory.go`.

In the second step the driver sends out as many of our packets as possible which is conceptually similar to receiving: save the virtual address for the cleaning process, get the descriptor and mark it for sending by writing the physical address of the packet, flags and size to it. Afterwards we pass control of the packet buffers to the NIC by advancing the tail register, again only once per batch to increase performance.

```
1  //second half of TxBatch(); shortened for the sake of briefness, refer to source code for
       ↪ detailed explanation
2  var sent uint32
3  for sent = 0; sent < numBufs; sent++ {
4          nextIndex := wrapRing(curIndex, queue.numEntries)
5          if cleanIndex == nextIndex {
6  break
7  }
8          buf := bufs[sent]
9          queue.virtualAddresses[curIndex] = buf
10         queue.txIndex = wrapRing(queue.txIndex, queue.numEntries)
11         txd := queue.descriptors[curIndex]
12         if isBig {
13                 size := binary.BigEndian.Uint32(buf.Pkt[20:24])
14                 binary.BigEndian.PutUint64(txd.raw[:8], binary.BigEndian.Uint64(buf.Pkt[:8])+64)
15                 binary.BigEndian.PutUint32(txd.raw[8:12], IXGBE_ADVTXD_DCMD_EOP|
                       ↪ IXGBE_ADVTXD_DCMD_RS|IXGBE_ADVTXD_DCMD_IFCS|IXGBE_ADVTXD_DCMD_DEXT|
                       ↪ IXGBE_ADVTXD_DTYP_DATA|size)
16                 binary.BigEndian.PutUint32(txd.raw[12:16], size<<IXGBE_ADVTXD_PAYLEN_SHIFT)
17         } else {
18                 //same operations with binary.LittleEndian
19         }
20         curIndex = nextIndex
21 }
22 setCReg32(dev.addr, IXGBE_TDT(int(queueID)), uint32(queue.txIndex))
23 return sent
```

## 4.7 Driver API

Ixy is designed to be simple and educative and as such we also favour a simple over an extensive API. It is designed similar to DPDKs API that builds on explicit memory allocation and is the same when it comes to sending and receiving packets and allocating memory. Our API ist also not hard to use. For demonstration purposes we included a simple forwarding application is included in the GitHub repository (see `ixy-fwd.go`) which has less then 80 lines and even less lines of actual Go code.

Initialization is done with a single invocation of the `IxyInit()` function thus minimizing any opportunity for errors from the side of the user. The receive process can also be done with a single function call of [IxgbeDevice].RxBatch() which takes a queue index and a slice of pointers to packet buffers which it then fills. Note that the length of the

slice is simultaneously the desired batch size. RxBatch() returns the number of packets received which can be dropped but in applications such as forwarding is usually relevant for the following steps. It should also be kept in mind that the application is responsible for polling with RxBatch() at the desired frequency.

Sending packets can also be done in a single function call of `[IxgbeDevice].TxBatch()` with the queue index and a slice of valid allocated packet buffers. This would be the case when forwarding packets as one would simply use the packets received from the `RxBatch()` but could also be manually allocated in other scenarios. `TxBatch()` returns the number of sent packets. One should carefully consider how to proceed with unsent packets; they can be resubmitted or dropped by freeing them but have to be returned to the mempool in one way or another. An application that wants to send its own packets instead of reusing received ones, such as a packet generator, will need to generate its own packets. A common approach is to first fill a mempool completely with a packet template and then free them all. This way each newly allocated packet will have this template which can then be altered according to ones needs such as updating sequence numbers and adding the calculated checksum. We implemented a packet generator (`ixy-pktgen.go`) which unfortunately does not work at the time of this thesis as calling `IxyInit()` currently seems to overwrites the mempool IDs in the packets that are previously allocated in `initMempool()` for no apparent reason.

An application might want to provide stats about the NIC. The `stats.go` file provides basic functionality for calculating stats for RX and TX queues. Stats are polled from the NIC through the use of `ReadStats()` in `ixgbe.go` and can then displayed using either `[DeviceStats].PrintStats()` or `[DeviceStats].PrintStatsDiff()`.

# Chapter 5

# Performance Analysis

Ixy was created as an educational driver and thus had simplicity as its highest priority. Nevertheless we tried to make the driver as fast as possible wherever we could and the cost in complexity was not too high. As the ixy reimplementations are rewrites of the original driver, this also allows us to evaluate the viability of different languages in the aspect of network driver programming. In this thesis we will compare our Go implementation with the original C driver. Since their features and architectures are mostly equal, this will provide a comparable performance evaluation between Go and C and the impact the choice of language has on the driver.

For this we compare the forwarding applications of both implementations. All measurements are made with Go version 1.11.1. A forwarder receives packets on up to two devices and changes a single byte in order to simulate a more realistic workload as this ensures that the packet data is fetched into the L1 cache. While one could use a dual port NIC as well, all our measurements are taken using two separate NICs as this eliminates possible hardware limits as described in the original paper [8]. A received packet on either device is then forwarded on the other. Neither the C nor the Go driver implement multi-threading. For our measurement we send packets with MoonGen [6] from one machine on both NICs for bidirectional forwarding and run the forwarding application on another machine which receives packets on two NICs. The packets have a payload of 60 bytes. The forwarding application receives the packets as described in section 4.6 with the `RxBatch()` function and sends them back out via use of `TxBatch()`, which also cleans the packet buffers in batches.

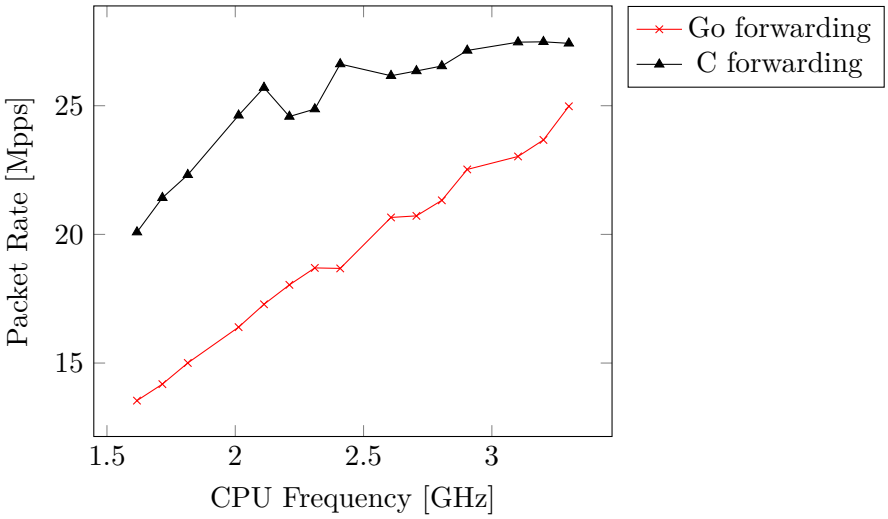



Figure 5.1: Single-core forwarding performance with varying CPU speed. Go batch size 256. C measurements taken from the original driver, CPU frequency measured with 2 separate NICs [7]

## 5.1 Benchmarks

We benchmarked the driver with the afore mentioned setup.

### 5.1.1 Throughput

In order to quantify baseline performance and identify bottlenecks, we run the forwarding example as explained above while increasing CPU frequency and compare them to the results of the C driver. The default CPU frequency at 100% is 3.3GHz, we start at 49% speed, 1.6GHz, and increase to the maximum frequency within the supported steps. The measurements were performed with a constant batch size of 256, where ixy.go is the fastest, and are presented in Figure 5.1. Note that we always calculate the average using the sum of both forwarding streams. Go ixy is significantly slower then C ixy at lower CPU frequencies but scales better so that both end up very close together with ixy.go being approximately 10% slower then the C implementation under optimal circumstances. Note that the slight CPU abnormality of the C implementation, where it is actually slightly faster at slighly below 100% speed, does not happen with Go which runs the fastest at full CPU frequency. As ixy.go ends up very close to the performance of the original ixy, this refutes many of the potential detriments concerning Go as a language for driver programming.

We touch the packets by changing one byte in order to simulate a more realistic workload. Nevertheless it is important to know how much it actually impacts performance.

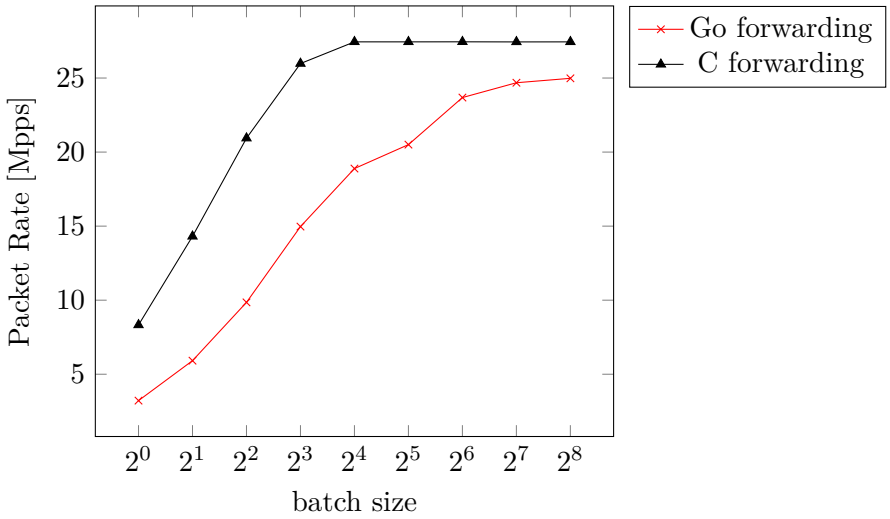



Figure 5.2: Single-core forwarding performance with varying batch size and 3.3GHz CPU frequency for all measurements. C measurements are the official performance measurements [7]

When measured at a batch size of 256 and at 3.3GHz and not touching the packets, ixy.go has an average throughput of 27.44 million packets per second which translates to almost a 10% increase in speed, bringing it up to the level of ixy.c. Meanwhile disabling packet touching in the C implementation actually has a negative effect on the throughput, dropping the average to 20.18 million packets per second though this is not further elaborated in the corresponding paper [8].

### 5.1.2 Batching

As explained in section 4.6, batching is essential for performance as updating the driver after operations requires access to the queue index register, a rather costly operation. Batching reduces the need for this register access to once per batch thus significantly reducing overhead. Figure 5.2 shows how the performance increases with higher batch sizes. While performance increases starkly within low batch sizes, it does so with diminishing return and even starts to drop of after a certain threshold. Ixys default batch size has been chosen as 32 as a result, the Go implementation starts out slower at lower batch sizes but catches up later until it peaks at a batch size of 256. The explanation for the diminishing returns is the same as it is for ixy: a higher batch size thrashes the cache and thus results in an increased number of cache misses [8] though in the case of ixy.go higher batch sizes still outweigh the penalties.

| Function | flat | flat% | cum | cum% |
|---|---|---|---|---|
| ixy.go/driver.(*ixgbeDevice).RxBatch | 8.86s | 28.97% | 12.18s | 39.83% |
| main.forward | 6.48s | 21.19% | 30.45s | 99.57% |
| ixy.go/driver.(*ixgbeDevice).TxBatch | 6.11s | 19.98% | 11.79s | 38.55% |
| encoding/binary.littleEndian.PutUint32 (inline) | 0.58s | 1.90% | 93.17% | 0.58s |
| 1.90% | | | | |
| encoding/binary.littleEndian.PutUint64 (inline) | 0.56s | 1.83% | 95.00% | 0.56s |
| 1.83% | | | | |
| encoding/binary.littleEndian.Uint64 (inline) | 0.26s | 0.85% | 96.83% | 0.26s |
| 0.85% | | | | |
| encoding/binary.littleEndian.Uint32 (inline) | 0.23s | 0.75% | 98.40% | 0.23s |
| 0.75% | | | | |

Table 5.1: CPU pprof shortened results

## 5.2 Profiling

Go offers powerful means of profiling with the pprof tool and package. In order to use this profiling on a standalone program, we have to add a minor amount of code to start and stop CPU profiling and write the profiles to files. Refer to the instructions in the pprof package [20] on how to enable profiling in standalone programs. pprof is a sampling profiler, meaning it halts execution a default of 100 times per second to record a sample of program counters and the current goroutine on the stack. We performed CPU and memory profiling on a forwarding program that ran for approximately 30 seconds after startup and took a look at the invocation graphs.

All following profiling is done with a batch size of 256. Table 5.1 shows where CPU time is spent on average, refer to the appendix for the full list. `flat` refers to the amount of time the method was actually running while `cum` records when the method was found on the stack while sampling. More interesting though is the invocation graph that pprof can provide. Time is split rather evenly between RxBatch and TxBatch, striking the balance needed for controlled forwarding. As one of the main concerns regarding Go was the potential detriment of the garbage collection on the speed, it is pleasant that it has next to no impact on the driver. The pprof tool does not list the garbage collection or any other elements of the runtime, except for the utility needed for cgo and thus is explicitly called, as they are dropped due to insignificance (total of <=0.15s for 64 nodes in 30.58s total measurement time). Thus we can conclude that the Go runtime is optimized to a degree where it is effectively a non factor. Lastly the put and get function from the binary package [19] consume 5.01% of the total time. While one could try to use unsafe operations to potentially speed up the driver by a few more

packets per second, we expect potential gains to be rather small and thus stick with the safe and idiomatic aproach. This consideration is detailed below in Section 5.3.

Go also offers memory profiling and we found that the results are very close to expectations. Memory is allocated during initialization and nowhere else except for profiling and a package from the standart library, meaning that there are no unecessary allocations, memory leaks and the like. The memory allocation graph is presented in figure 5.4.

## 5.3 Go Specific Performance Considerations

We have now seen that ixy.go, while slightly behind in terms of packet processing ability, can definitely compete with ixy.c. While there is always room for improvement, one has to be aware of its costs which often come in the form of complexity.

### 5.3.1 Unsafe Pointer Operations

We originally considered to try and use unsafe pointers instead of the provided byte slices for the DMA memory. Our current approach accesses these as is through the use of the encoding/binary package. The pprof tool shows that encoding/binary.LittleEndian.* functions are automatically inlined with a total of 5.01% of the applications run time. We use these functions as we need a way to address the byte slice that we get as a result from the mmap() syscall. Another option to this would be a C-like approach by casting the slice to unsafe pointers to the same memory area and access it directly as pointers instead of invoking the functions from the binary package. However in light of the frequency the fuction is used, we doubt that this approach would significantly increase speed. Additionally, as mentioned before, we try to keep the use of unsafe operations to a minimum wherever possible as it defeats the type system and bypasses some of Go's core features which arises the question wh one should even use Go then.

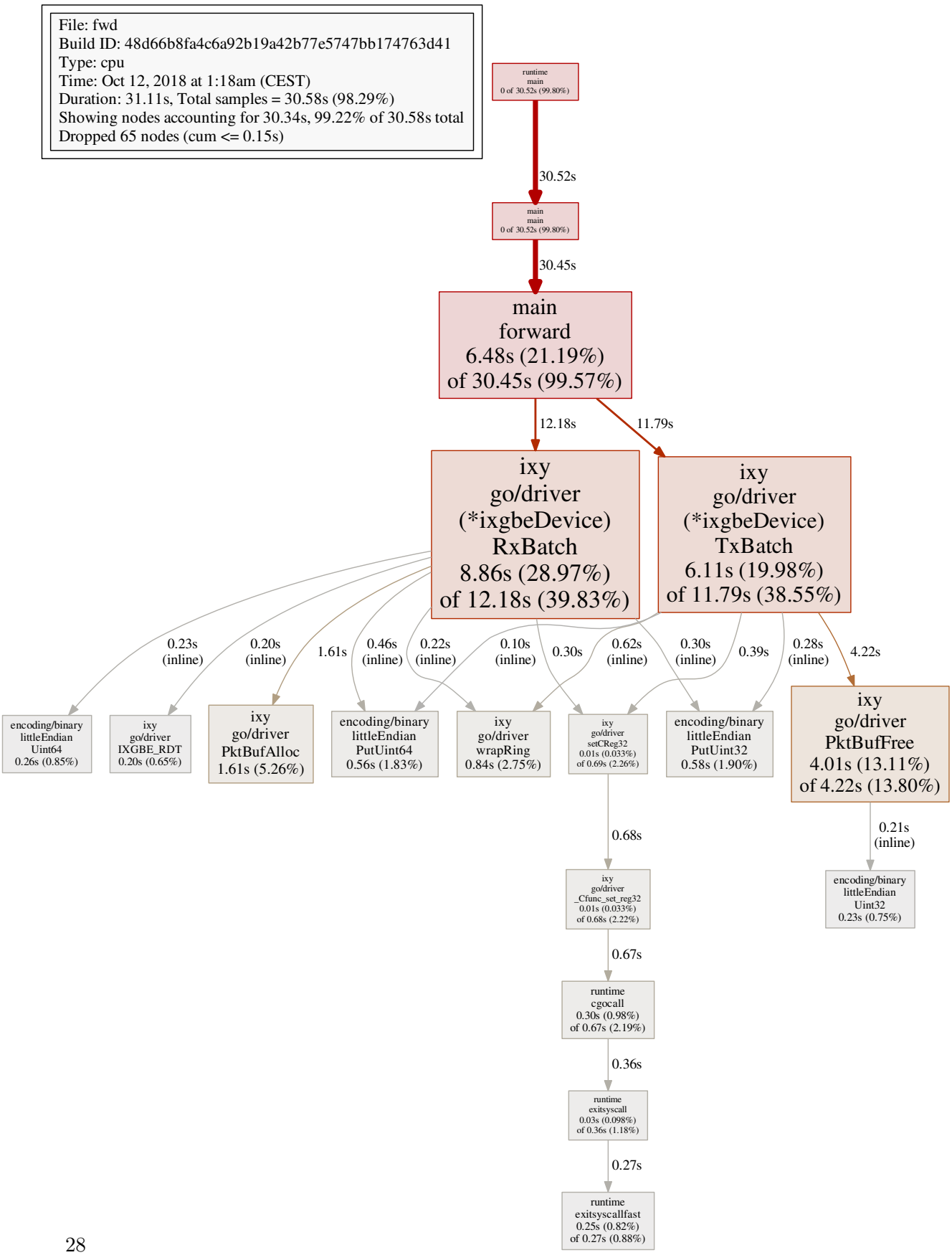



Figure 5.3: CPU profiling graph

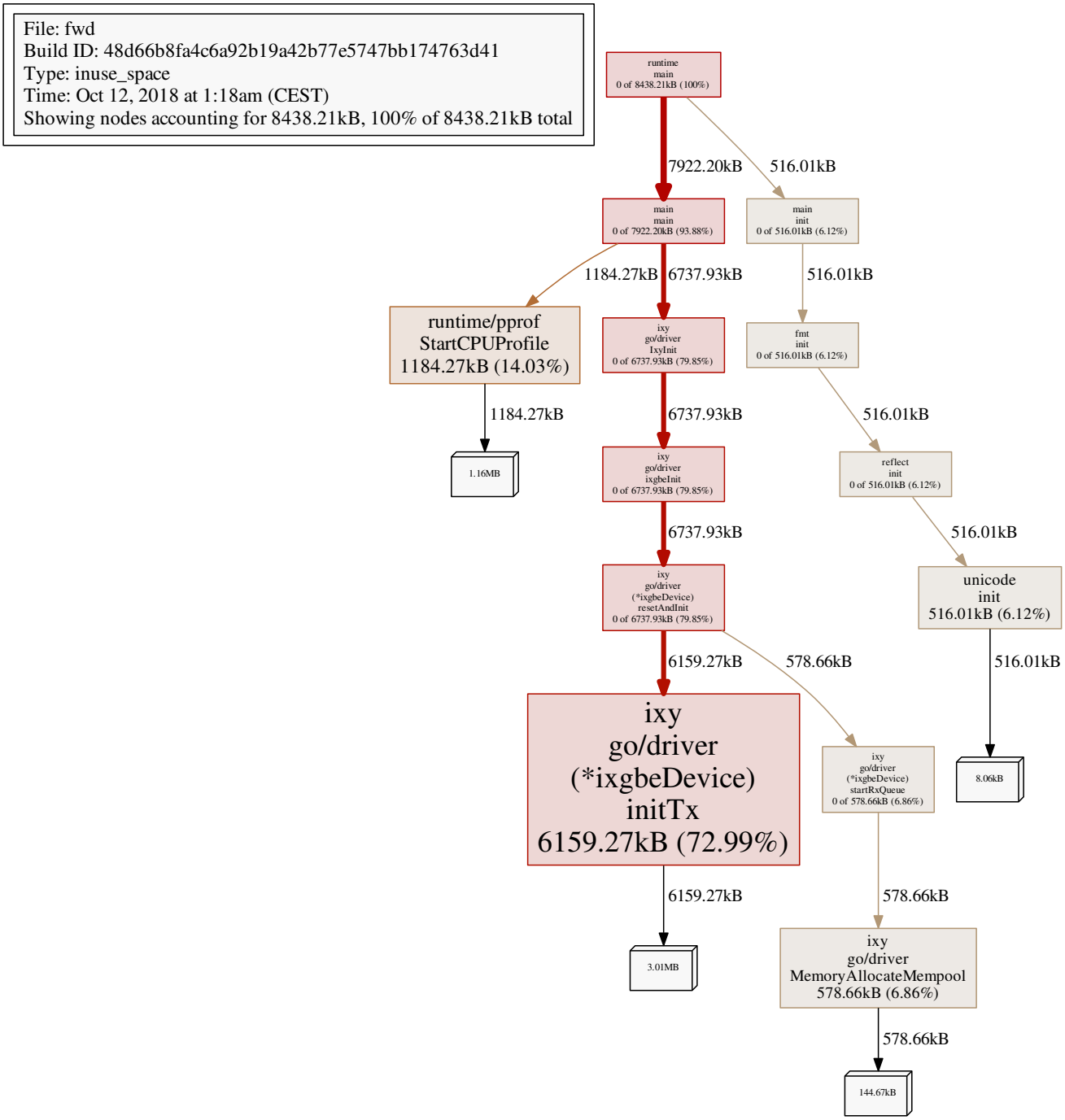



Figure 5.4: Memory profiling graph

# Chapter 6

# Evaluation

We have presented an architectural overview of ixy.go and measured performance. Now we will try to answer the question of whether Go is a suitable programming language for (network) driver development and evaluate its advantages and disadvantages in this area with regards to the characteristics listed in section 2. The main points of discussion for this topic will be performance, development and suitability for educational purposes. While these are all important aspects in the context of education, a real world production grade network driver's only relevant category is speed, as long as there are no security issues. We will then draw the final conclusions concerning Go as a language for network drivers based on our findings.

## 6.1 Performance

Performance is *the* most central point of consideration when choosing a language for a network driver and thus only languages that can produce code that performs at least as good as C code can be considered alternatives. Should a language provide ample other benefits, one can potentially accept a slight loss of performance. Languages that cannot provide code with similar speed are not suited for production grade drivers and are unusable in high performance environments.

In our performance analysis we evaluated our driver in regards of CPU frequency and batch size. While C performed better under unfavourable conditions, Go is only ≈10% slower when both applications are being run under optimal conditions. While this might not be a huge gap, speed is usually the single most important aspect of network drivers and as such slower drivers rarely find use. Nevertheless this makes Go still a candidate

when choosing the language to write a network driver with. Most drivers for Linux to this date are written in C without even considering other languages and while C is mandatory for kernel programming, this fact does not change in the realm of user space drivers. We evaluated the advantages and disadvantages of Go at length in Chapter 2. Should one plan to implement a new network driver which does not have speed as its uncontested top priority, Go is a reasonable choice as it offers a wide array of additional features ranging from security aspects to an extensive tool chain while still running at a reasonable speed.

## 6.2 Development

Motivation for the development of this port was that on the one hand it furthers accessibility and thus can teach a broader audience, and on the other hand to explore the possibilities of modern programming languages for driver development. In this section we will focus on the latter. Compared to C, Go offers many security features, a vast standard library, more modern syntax, a vast array of tools and many other things which make for a strong case. We will now discuss their influence on the development process compared to what it would have been in C as well as how it differs from typical Go programs.

Starting at the outermost scope of inspection, we see that the codebase of the drivers is roughly equal, with Go being a little larger. While the C implementation has slightly less then 1000 lines, the Go implementation comprises ≈1000 lines of code for the driver and another ≈70 lines of code for the forwarding application, excluding the machine readable versions of the datasheet. This difference largely stems from the differences in our way of addressing DMA memory, the calls to C and the fact that the linter, the formatting tool that enforces the Go coding style, tends to spread some statements that could be expressed in a single line over multiple lines instead. While this helps readability it also somewhat inflates the line number. Subtracting these inflations would bring us almost exactly to C level.

As discussed in section 2, Go orients itself style wise at the C programming language but tries to modernise many approaches that were deemed outdated. This, combined with syntactic sugar and a type inference system, makes for an easier, safer and more intuitive development and often helps readability as well. We present two code examples comparing the same function snippets from ixy and ixy.go.

ixy_tx_batch_busy_wait from device.h

```
static void ixy_tx_batch_busy_wait(struct ixy_device* dev, uint16_t queue_id, struct pkt_buf*
    ↪ bufs[], uint32_t num_bufs) {
	uint32_t num_sent = 0;
	while ((num_sent += ixy_tx_batch(dev, 0, bufs + num_sent, num_bufs - num_sent)) !=
	    ↪ num_bufs) {
	}
}
```

IxyTxBatchBusyWait from device.go

```
func IxyTxBatchBusyWait(dev IxyInterface, queueID uint16, bufs []*PktBuf) {
	numBufs := uint32(len(bufs))
	for numSent := uint32(0); numSent != numBufs; numSent += dev.TxBatch(queueID, bufs[numSent
	    ↪ :]) {
	}
}
```

start_rx_queue from ixgbe.c

```
static void start_rx_queue(struct ixgbe_device* dev, int queue_id) {
	struct ixgbe_rx_queue* queue = ((struct ixgbe_rx_queue*)(dev->rx_queues)) + queue_id;
	uint32_t mempool_size = NUM_RX_QUEUE_ENTRIES + NUM_TX_QUEUE_ENTRIES;
	queue->mempool = memory_allocate_mempool(mempool_size < 4096 ? 4096 : mempool_size, 2048);
	if (queue->num_entries & (queue->num_entries - 1)) {
		error("number␣of␣queue␣entries␣must␣be␣a␣power␣of␣2");
	}
	for (int i = 0; i < queue->num_entries; i++) {
		volatile union ixgbe_adv_rx_desc* rxd = queue->descriptors + i;
		struct pkt_buf* buf = pkt_buf_alloc(queue->mempool);
		if (!buf) {
			error("failed␣to␣allocate␣rx␣descriptor");
		}
		rxd->read.pkt_addr = buf->buf_addr_phy + offsetof(struct pkt_buf, data);
		rxd->read.hdr_addr = 0;
		queue->virtual_addresses[i] = buf;
	}
	//set registers
}
```

startRxQueue from ixgbe.go

```
func (dev *ixgbeDevice) startRxQueue(queueID int) {
	queue := &dev.rxQueues[queueID]
	mempoolEntries := uint32(numRxQueueEntries + numTxQueueEntries)
	if mempoolEntries < 4096 {
		mempoolEntries = 4096
	}
	queue.mempool = MemoryAllocateMempool(mempoolEntries, 2048)
	if queue.numEntries&(queue.numEntries-1) != 0 {
		log.Fatal("number␣of␣queue␣entries␣must␣be␣a␣power␣of␣2")
	}
	for i := uint16(0); i < queue.numEntries; i++ {
		rxd := queue.descriptors[i]
		buf := PktBufAlloc(queue.mempool)
		if buf == nil {
			log.Fatal("failed␣to␣allocate␣rx␣descriptor")
		}
		if isBig {
			binary.BigEndian.PutUint64(rxd.raw[:8], binary.BigEndian.Uint64(buf.Pkt[:8])
			    ↪ +64)
```

```
19                      binary.BigEndian.PutUint64(rxd.raw[8:], uint64(0))
20              } else {
21                      binary.LittleEndian.PutUint64(rxd.raw[:8], binary.LittleEndian.Uint64(buf.
                            ↪ Pkt[:8])+64)
22                      binary.LittleEndian.PutUint64(rxd.raw[8:], uint64(0))
23              }
24              queue.virtualAddresses[i] = buf
25      }
26      //set registers
27  }
```

(These examples are edited for the sake of brevity and formatting. Refer to the respective source code for more details.)

The first example of the busy waiting shows quite a few differences in terms of syntax. Go functions are declared with the func keyword and as per convention are written in camel case. Similar to the afore mentioned "name before type", functions are declared in the order of [function name]([arguments]) [return vales] {[function body]}. Compared to the spiral ordering of C, this makes functions easy to read from left to right without interrupts. Furthermore we can see that a length as parameter is not necessary as bound checking is an integral part of Go and can be easily accessed. Variable declaration can be done via type inference using the ":=" operator and notably Go does not have a while statement as for can be used fully equally to it. Also, contrary to C, it is not possible to mix assignments into conditional statements.

In the second example we can observe a few more peculiarities of the Go programming language. We can define receiver functions by defining a type the method has to be invoked on. Next it has to be noted that Go does not support the convenient, but technically unnecessary "x?y:z" operator. Lastly in order to access fields of a struct pointer no "->" is needed, dotted notation works in Go for direct as well as pointer access. Most other constructs are the result of different data structures which mostly makes things more readable but requires longer statements when accessing DMA memory as we have to translate to and from byte slices.

The increased clarity as well as the more expressive and easier to read syntax may not directly reduce bugs, errors and the like but does have notable positive effects. Simpler syntax makes it easier to detect errors and has and even larger impact on projects many people work on, especially those of a larger scale, as it can avoid misunderstandings, reduces the time needed to understand code written by others and makes in generally more maintainable.

Additionally Go offers features C does not that can enhance the quality of the project in many ways such as expressiveness, safety and expandability. A notable example is the interface we used for the drivers and while we only support ixgbe currently, additional

support for VirtIO and the like requires nothing more then implementing the interface methods and adding it as part of the driver package. This approach is much cleaner and expandable then the function pointers of the C driver. But Go has much more to offer then what was used for the driver. Go offers native support for concurrency in the form of goroutines which are designed to be clean and easy to handle while simultaneously efficient in terms of thread usage as well as concurrency.

We showed the results obtained via the pprof tool in section 5.2 which aided a lot with optimization. As an example an earlier version ran at approximately 8 million packets per second, a third of our current speed. The reason for this was that we didn't reuse packet buffers and instead always allocated a new one which, albeit little more then a struct of essentially two pointers, adds up tremendously. The graph produced by the pprof tool (see Appendix A.1) very clearly showed that we used most of the time during packet allocation and upon further inspection one can identify runtime.newobject as the main culprit. From there on it is simple to search the code for object allocations.

## 6.3 Suitability for Low-Level Tasks

One of the biggest problem during development was low-level memory management. Specifically register access has proven itself to be difficult in Go. To recapitulate: register access happens though the use of the BAR0 address and the correct offsets. Whether read or write, we need a compiler barrier. In C this can be solved with a so called memory barrier, essential to this is the keyword volatile which marks the bit of code to not be optimized by the compiler. Go does not support volatile though it offers atomic reads and writes through the atomic package. For this to work, we would have to make use of the unsafe package to essentially convert the subslice into a pointer. As we wanted to keep unsafe code to a minimum, we instead chose to employ the C code from the original driver as an opportunity to present cgo.

On the other hand we were surprised about the garbage collection. Originally named as the reason why Go is nor suited for systems programming, our analysis has proven otherwise. As all our memory allocation is done in bulk during the initialization process and not allocations during the forwarding process, the garbage collector has nothing to clean and little reason to run often which results in it not even being listed in the pprof report due to insignificance. We can therefore conclude that Go is indeed suited for low-level tasks, provided that memory is handled reasonably.

## 6.4 Educational Purposes

Go is an interesting choice of language for educational purposes. It tends to be easier to read and does not require much understanding of the language itself in order to understand the code, especially compared to some C constructs like function pointer, pointer casting and other more intricate operations. It thus aids the purpose of being an educational driver as it lowers entry barriers and demands less attention to the code instead of the actual workings of a network driver.

## 6.5 Conclusion

Now that we have evaluated the Go implementation of the ixy driver from multiple perspectives, it is time to come to a conclusion:
Should you use Go as a language for driver programming? The answer depends on the requirements.

Go was primarily designed for networking and server programming, as these are the core business areas of Google. As seen in Chapter 3, even the network drivers for the zircon microkernel, developed by Google, do not use Go for this purpose but instead stick to C and C++. Biscuit has proven, that one can indeed use Go as a systems programming language, albeit at the cost of performance. While the performance penalty was only calculated for the kernel as a whole, the numbers roughly match our findings concerning a network driver written in Go.

Go is indeed somewhat slower then C for the purpose of network drivers, but when speed is not the highest priority it does become a solid option due to improved safety, increased readability, a runtime that provides an array of critical features, tools that aid in development and a vast standard library as well as many other libraries [9].

## 6.6 Outlook

In this thesis we implemented and evaluated a simple network driver in the Go programming language. Nevertheless there is still much room for potential future work. First of all the IxyInterface offers an easy way for the support of additional drivers such as the VirtIO driver that is also implemented in the original work. Second we evaluated the driver on a rather basic level, mainly looking at throughput in order to analyse performance, though it is not the only factor for determining what makes a driver good

as other properties such as latency are also important factors. This is especially interesting in the face of the next point: writing higher level applications on top of ixy.go and evaluating the impact of different packet sizes and packet processing. And as a last point it would be interesting to implement a highly optimized version of the driver for maximised speed akin to DPDK [13] in Go.

# Chapter A

# Appendix

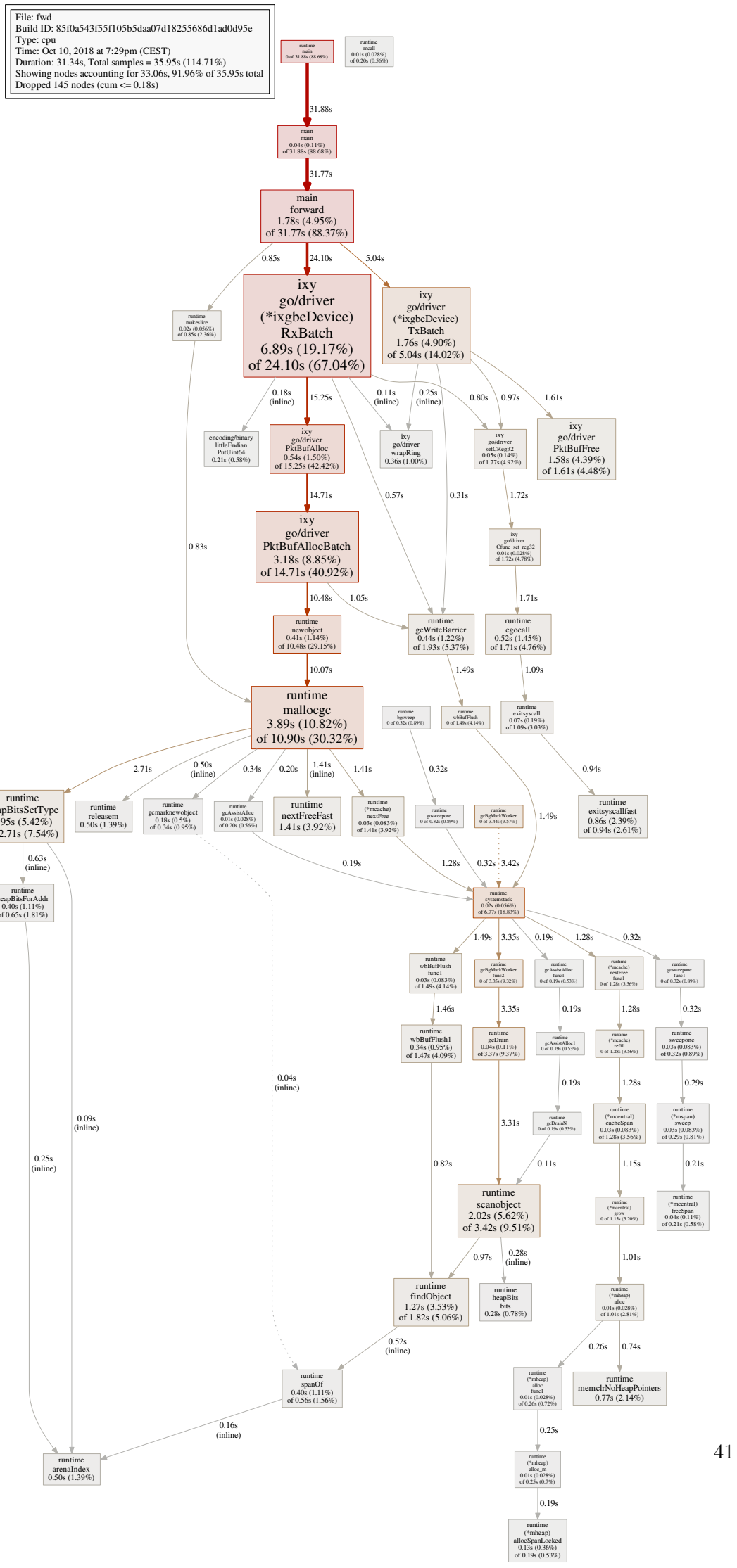
CPU profiling of all nodes that were not dropped:

```
(pprof) top18
Showing nodes accounting for 30.34s, 99.22% of 30.58s total
Dropped 65 nodes (cum <= 0.15s)
      flat  flat%   sum%        cum   cum%
     8.86s 28.97% 28.97%     12.18s 39.83%  ixy.go/driver.(*ixgbeDevice).RxBatch
     6.48s 21.19% 50.16%     30.45s 99.57%  main.forward
     6.11s 19.98% 70.14%     11.79s 38.55%  ixy.go/driver.(*ixgbeDevice).TxBatch
     4.01s 13.11% 83.26%      4.22s 13.80%  ixy.go/driver.PktBufFree
     1.61s  5.26% 88.52%      1.61s  5.26%  ixy.go/driver.PktBufAlloc
     0.84s  2.75% 91.27%      0.84s  2.75%  ixy.go/driver.wrapRing (inline)
     0.58s  1.90% 93.17%      0.58s  1.90%  encoding/binary.littleEndian.PutUint32 (inline)
     0.56s  1.83% 95.00%      0.56s  1.83%  encoding/binary.littleEndian.PutUint64 (inline)
     0.30s  0.98% 95.98%      0.67s  2.19%  runtime.cgocall
     0.26s  0.85% 96.83%      0.26s  0.85%  encoding/binary.littleEndian.Uint64 (inline)
     0.25s  0.82% 97.65%      0.27s  0.88%  runtime.exitsyscallfast
     0.23s  0.75% 98.40%      0.23s  0.75%  encoding/binary.littleEndian.Uint32 (inline)
     0.20s  0.65% 99.05%      0.20s  0.65%  ixy.go/driver.IXGBE_RDT (inline)
     0.03s 0.098% 99.15%      0.36s  1.18%  runtime.exitsyscall
     0.01s 0.033% 99.18%      0.68s  2.22%  ixy.go/driver._Cfunc_set_reg32
     0.01s 0.033% 99.22%      0.69s  2.26%  ixy.go/driver.setCReg32
         0     0% 99.22%     30.52s 99.80%  main.main
         0     0% 99.22%     30.52s 99.80%  runtime.main
```

Memory profiling of all nodes that were not dropped:

```
(pprof) top13
Showing nodes accounting for 8438.21kB, 100% of 8438.21kB total
      flat  flat%   sum%        cum   cum%
 6159.27kB 72.99% 72.99%  6159.27kB 72.99%  ixy.go/driver.(*ixgbeDevice).initTx
 1184.27kB 14.03% 87.03%  1184.27kB 14.03%  runtime/pprof.StartCPUProfile
  578.66kB  6.86% 93.88%   578.66kB  6.86%  ixy.go/driver.MemoryAllocateMempool
  516.01kB  6.12%   100%   516.01kB  6.12%  unicode.init
         0     0%   100%   516.01kB  6.12%  fmt.init
         0     0%   100%  6737.93kB 79.85%  ixy.go/driver.(*ixgbeDevice).resetAndInit
         0     0%   100%   578.66kB  6.86%  ixy.go/driver.(*ixgbeDevice).startRxQueue
         0     0%   100%  6737.93kB 79.85%  ixy.go/driver.IxyInit
         0     0%   100%  6737.93kB 79.85%  ixy.go/driver.ixgbeInit
         0     0%   100%   516.01kB  6.12%  main.init
```

```
14          0     0%   100%  7922.20kB 93.88%  main.main
15          0     0%   100%   516.01kB  6.12%  reflect.init
16          0     0%   100%  8438.21kB   100%  runtime.main
```


File: fwd
Build ID: 85f0a543f55f105b5daa07d18255686d1ad0d95e
Type: cpu
Time: Oct 10, 2018 at 7:29pm (CEST)
Duration: 31.34s, Total samples = 35.95s (114.71%)
Showing nodes accounting for 33.06s, 91.96% of 35.95s total
Dropped 145 nodes (cum <= 0.18s)
runtime main 0 of 31.88s (88.68%)
runtime mcall 0.01s (0.028%) of 0.20s (0.56%)
31.88s
main main 0.04s (0.11%) of 31.88s (88.68%)
31.77s
main forward 1.78s (4.95%) of 31.77s (88.37%)
0.85s
24.10s
5.04s
runtime makeslice 0.02s (0.056%) of 0.85s (2.36%)
ixy go/driver (*ixgbeDevice) RxBatch 6.89s (19.17%) of 24.10s (67.04%)
ixy go/driver (*ixgbeDevice) TxBatch 1.76s (4.90%) of 5.04s (14.02%)
0.18s (inline)
15.25s
0.11s (inline)
0.25s (inline)
0.80s
0.97s
1.61s
encoding/binary littleEndian PutUint64 0.21s (0.58%)
ixy go/driver PktBufAlloc 0.54s (1.50%) of 15.25s (42.42%)
ixy go/driver wrapRing 0.36s (1.00%)
ixy go/driver setCReg32 0.05s (0.14%) of 1.77s (4.92%)
ixy go/driver PktBufFree 1.58s (4.39%) of 1.61s (4.48%)
14.71s
0.57s
0.31s
1.72s
0.83s
ixy go/driver PktBufAllocBatch 3.18s (8.85%) of 14.71s (40.92%)
ixy go/driver _Cfunc_set_reg32 0.01s (0.028%) of 1.72s (4.78%)
10.48s
1.05s
1.71s
runtime newobject 0.41s (1.14%) of 10.48s (29.15%)
runtime gcWriteBarrier 0.44s (1.22%) of 1.93s (5.37%)
runtime cgocall 0.52s (1.45%) of 1.71s (4.76%)
10.07s
1.49s
1.09s
runtime mallocgc 3.89s (10.82%) of 10.90s (30.32%)
runtime bgsweep 0 of 0.32s (0.89%)
runtime wbBufFlush 0 of 1.49s (4.14%)
runtime exitsyscall 0.07s (0.19%) of 1.09s (3.03%)
2.71s
0.50s (inline)
0.34s
0.20s
1.41s (inline)
1.41s
0.32s
0.94s
runtime heapBitsSetType 1.95s (5.42%) of 2.71s (7.54%)
runtime releasem 0.50s (1.39%)
runtime gcmarknewobject 0.18s (0.5%) of 0.34s (0.95%)
runtime gcAssistAlloc 0.01s (0.028%) of 0.20s (0.56%)
runtime nextFreeFast 1.41s (3.92%)
runtime (*mcache) nextFree 0.03s (0.083%) of 1.41s (3.92%)
runtime gosweepone 0 of 0.32s (0.89%)
runtime gcBgMarkWorker 0 of 3.44s (9.57%)
1.49s
runtime exitsyscallfast 0.86s (2.39%) of 0.94s (2.61%)
0.63s (inline)
0.19s
1.28s
0.32s
3.42s
runtime heapBitsForAddr 0.40s (1.11%) of 0.65s (1.81%)
runtime systemstack 0.02s (0.056%) of 6.77s (18.83%)
1.49s
3.35s
0.19s
1.28s
0.32s
runtime wbBufFlush func1 0.03s (0.083%) of 1.49s (4.14%)
runtime gcBgMarkWorker func2 0 of 3.35s (9.32%)
runtime gcAssistAlloc func1 0 of 0.19s (0.53%)
runtime (*mcache) nextFree func1 0 of 1.28s (3.56%)
runtime gosweepone func1 0 of 0.32s (0.89%)
1.46s
3.35s
0.19s
1.28s
0.32s
runtime wbBufFlush1 0.34s (0.95%) of 1.47s (4.09%)
runtime gcDrain 0.04s (0.11%) of 3.37s (9.37%)
runtime gcAssistAlloc1 0 of 0.19s (0.53%)
runtime (*mcache) refill 0 of 1.28s (3.56%)
runtime sweepone 0.03s (0.083%) of 0.32s (0.89%)
0.04s (inline)
0.19s
1.28s
0.29s
0.09s (inline)
runtime gcDrainN 0 of 0.19s (0.53%)
runtime (*mcentral) cacheSpan 0.03s (0.083%) of 1.28s (3.56%)
runtime (*mspan) sweep 0.03s (0.083%) of 0.29s (0.81%)
3.31s
0.25s (inline)
0.82s
0.11s
1.15s
0.21s
runtime scanobject 2.02s (5.62%) of 3.42s (9.51%)
runtime (*mcentral) grow 0 of 1.15s (3.20%)
runtime (*mcentral) freeSpan 0.04s (0.11%) of 0.21s (0.58%)
0.97s
0.28s (inline)
1.01s
runtime findObject 1.27s (3.53%) of 1.82s (5.06%)
runtime heapBits bits 0.28s (0.78%)
runtime (*mheap) alloc 0.01s (0.028%) of 1.01s (2.81%)
0.52s (inline)
0.26s
0.74s
runtime spanOf 0.40s (1.11%) of 0.56s (1.56%)
runtime (*mheap) alloc func1 0.01s (0.028%) of 0.26s (0.72%)
runtime memclrNoHeapPointers 0.77s (2.14%)
0.16s (inline)
0.25s
runtime arenaIndex 0.50s (1.39%)
runtime (*mheap) alloc_m 0.01s (0.028%) of 0.25s (0.7%)
0.19s
runtime (*mheap) allocSpanLocked 0.13s (0.36%) of 0.19s (0.53%)

# Chapter B

## List of acronyms

| | |
|---|---|
| API | Application Programming Interface. A set of tools for buildung software. |
| BAR | Base Address Register. |
| CSP | Communicating Sequential Processes Formal language for describing patterns of interaction in concurrent systems. |
| DPDK | Data Plane Development Kit. User space network driver originally developed by Intel and handed over to the Linux Foundation. |
| NIC | Network Interface Controller. |
| PCI | Application Programming Interface. Local bus for attaching hardware to a computer. |

# Bibliography


[1] Luke Gorrie et al. *Snabb: Simple and fast packet networking*. `https://github.com/snabbco/snabb`. Last accessed on 2018/10/01.

[2] Kyle Bradshaw. *Fuchsia Friday: The Four Layers of Fuchsia*. `https://9to5google.com/2018/03/16/fuchsia-friday-the-four-layers-of-fuchsia/`. Last accessed on 2018/10/03.

[3] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. "The benefits and costs of writing a POSIX kernel in a high-level language". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 89–105. ISBN: 978-1-931971-47-8. URL: `https://www.usenix.org/conference/osdi18/presentation/cutler`.

[4] Linux Kernel Documentation. *Page Migration*. `https://www.kernel.org/doc/Documentation/vm/page_migration`. Last accessed on 2018/10/01.

[5] Paul Emmerich et al. *ixy, a simple userspace packet processing for educational purposes*. Last accessed on 2018/10/03.

[6] Paul Emmerich et al. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015.

[7] Paul Emmerich et al. *Performance measurements of ixy used in the paper*. Last accessed on 2018/10/08.

[8] Paul Emmerich et al. *User Space Network Drivers*. Last accessed on 2018/10/03.

[9] *Go libraries*. `https://golanglibs.com/`. Last accessed on 2018/10/12.

[10] Rhys Hiltner. *Go's march to low-latency GC*. `https://blog.twitch.tv/gos-march-to-low-latency-gc-a6fa96f06eb7`. Last accessed on 2018/10/01.

[11] Intel. *Intel 82599 10 GbE Controller Datasheet Rev. 3.3*.

[12] *Netstack: IPv4 and IPv6 userland network stack*. `https://github.com/google/netstack/`. Last accessed on 2018/10/03.

[13] DPDK Project. *DPDK Website*. `http://dpdk.org`. Last accessed on 2018/10/01.

[14] Fuchsia Project. *Garnet Repository*. `https://fuchsia.googlesource.com/garnet`. Last accessed on 2018/10/03.

[15] Fuchsia Project. *Zircon Kernel, Core Drivers and Services*. `https://fuchsia.googlesource.com/zircon/`. Last accessed on 2018/10/03.

[16] Go Project. *Effective Go*. `https://golang.org/doc/effective_go.html`. Last accessed on 2018/10/01.

[17] Go Project. *Go 1 and the Future of Go Programs*. `https://golang.org/doc/go1compat`. Last accessed on 2018/10/01.

[18] Go Project. *Go: Frequently Asked Questions*. `https://golang.org/doc/faq`. Last accessed on 2018/10/01.

[19] Go Project. *Package binary*. `https://golang.org/pkg/encoding/binary/`. Last accessed on 2018/10/01.

[20] Go Project. *Package pprof*. `https://golang.org/pkg/runtime/pprof/`. Last accessed on 2018/10/08.

[21] Go Project. *Package syscall*. `https://golang.org/pkg/syscall/`. Last accessed on 2018/10/01.

[22] Go Project. *The Go Programming Language Specification*. `https://golang.org/ref/spec`. Last accessed on 2018/10/01.

[23] Linux Kernel Project. *Linux kernel coding style*. `https://www.kernel.org/doc/Documentation/process/coding-style.rst`. Last accessed on 2018/10/01.

[24] Intel DPDK Validation Team. *DPDK Intel NIC Performance Report Release 18.02*. Last accessed on 2018/10/01. May 2018.