Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Writing Network Drivers in Haskell

Alex Egger

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Writing Network Drivers in Haskell

Netzwerktreiber in Haskell schreiben

Author:	Alex Egger
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich
Date:	February 15, 2018

Abstract

Drivers that run in user-space have lots of advantages over traditional kernel drivers. One of these advantages is the choice of programming language is not limited to the traditional few languages.

To understand whether Haskell is a viable choice to write a network driver in, we implement a user-space network driver for Intel 82599 network interface cards and evaluate its performance. The driver, named `ixy.hs` after the original implementation `ixy`, is around 1000 lines of Haskell code and implements basic functionality for receiving and sending packets on compatible NICs.

We evaluate the resulting driver by looking at its forwarding capabilities in a high throughput scenario, conclude whether Haskell can keep up with other languages in the implementation of user-space network drivers and whether it can aid the developer in the creation of a reliable driver.

Contents

Chapter 1

Introduction

Device drivers are arguably complex pieces of software whose requirements are very strict on reliability and performance. Functional languages can provide a fair amount of intellectual advantage towards structuring complex applications well. A well-designed architecture leads to easier feature introduction, debugging and allows for increased re-usability of code [10]. At first glance this makes functional languages a prime target for the development of device drivers. In the following chapters we investigate which properties of functional languages are well suited for this task, and which properties may impact the development negatively. We then go on to inspect the `ixy` Haskell rewrite, `ixy.hs`, and see how it performs and finally answer the question: should one write network drivers in Haskell?

1.1 On Haskell & functional languages

Haskell is a polymorphically statically typed, lazy, purely functional language [12]. To understand these properties that make Haskell a language that is very near to some enthusiast's hearts, we must first look at some crude definitions.
According to Hughes what makes a language functional, as opposed to the traditional imperative pattern, is immutability of variables, a lack of side effects in functions and *referential transparency* [10]. Referential transparency means that a function always provides the same output for a certain input. The term lazy in the above definition of Haskell refers to a very specific aspect of the way expressions are evaluated in Haskell. It refers to lazy evaluation of expressions, which means expressions are only evaluated, when their actual value is needed and not at the time of assignment or when they are

passed to a function as a parameter [15]. Next we look at what makes Haskell a strong choice of language for any kind of programming work.

1.1.1 Abstraction

Two fundamental areas of programming are memory management and sequencing of instructions. Memory management has been largely abstracted away by some modern languages with the use of garbage collection. Other languages, e.g., Rust or partially D, still work with manual management, but there is a certain visible trend towards the compiler supporting the programmer in the memory management. Haskell is a garbage-collected language, so its memory management has also been largely abstracted away from the programmer's control. Imperative languages are based on sequencing instructions one after another and so defining a flow of execution, that will eventually provide some kind of desired result. In these languages abstracting away the sequencing from the programmer is virtually impossible, since the whole model of execution of the language relies on sequencing. Functional languages do not depend on an order of execution, rather the programmer defines a desired computation, not when or how it is executed. By doing this functional languages abstract away the component of sequencing from the programmer, just like some languages abstract away memory management with the use of garbage collection [20].

1.1.2 Type System

Haskell is a strongly typed language. This means in Haskell there are no implicit type conversions, as can be gathered by looking at `ixy.hs`'s code base. There are a total of 24 calls to `fromIntegral` to convert an integral type into a numerical one. As can be seen this results in quite a bit of work for the programmer, but it does provide the advantage of prohibiting unintentionally using a variable as a different type, which most probably could result in a bug. Additionally Haskell is statically type-checked, which means the types of variables are checked at compile-time, and not a run-time. This makes any kind of type error during run-time impossible. A few select languages including Standard ML, F, OCaml, Rust, Haskell and others also support automatic type inference. This means the compiler can without any user annotations infer the type of a variable and use this to type-check the variable at compile-time.

1.1.3 Garbage collection

Haskell is a garbage-collected language. It is also a language with (mostly) immutable data types. The garbage collector uses this fact to simplify and speed up its operation. Immutability forces a lot of temporary variables to be allocated, since variables can not be changed. Haskell's garbage collector scans only the last created set of variables

and marks those variables that are reachable from it as alive. The entire rest of the temporary variables can be removed by the garbage collector [8].

1.2 Drivers in Linux

There are two options on how to develop a driver, for a device of your choosing, on Linux.

1.2.1 Kernel space

The first, and more traditional, way is to write a kernel module, which can then be dynamically loaded by the kernel. These modules run in kernel space, entrusting them with control over the whole system, which requires special care from the developer of the module. An error in a module could jeopardize the integrity of the system, by e.g., writing to a wrong memory location and causing a crash of the system, or a corruption of data. This means a developer must be far more careful and thorough, when providing a kernel module to a user. Development of kernel modules is further complicated by a lack of debugging tools and limited ability to test execution. With kernel modules there is not much of a choice for which language to implement them in. Functions supplied by the kernel are written in C, which means language with tedious foreign function interface functionality can become a nightmare for the developer. Further the architecture of the driver system in the kernel is designed in a way, that simply can't harness the advantages other languages could provide. A driver developed in a different language would always have to accept having to bend their architecture to fit into the C architecture the kernel prescribes and could not really use the more advanced features the languages may bring. Additionally anecdotal evidence suggests a certain language preference concerning kernel code among kernel maintainers, which makes it unlikely that the developed kernel module would be included in the Linux kernel source code, if that were a goal of the efforts [16].

1.2.2 User space

The other option is to develop a user-space driver. User-space drivers run in user-space as opposed to running in kernel space, like a kernel module. This comes with a decrease in privileges, when compared to a kernel module, which means the likelihood to produce a drastic bug is reduced. A huge advantage of writing a user-space driver is the choice of language is essentially up to the developer. Since the driver does not rely on the functions provided by the kernel any language could be chosen theoretically. Another quality-of-life improvement is the developer can now again make use of his

debugging tools to debug certain sections of his driver. When using a kernel module whose functions are called from user space an overhead is produced due to context switches. User-space drivers do not suffer from this overhead, since they do not call functions in kernel space. This makes it possible to write even faster drivers, than a kernel module approach could. In fact a number of user-space driver frameworks have reached popularity in recent years, like Intel's DPDK [3] and Snabb [19].

1.2.3 Ixy and Ixy.hs

`Ixy` is a user-space driver for Intel NICs in the `ixgbe` family. Its purpose is to show that driving a user-space driver for a network card is not an impossible task. For this purpose it tries to be concise, especially readable, and free of external dependencies. It is written in C, as it provides a solid base to compare other language implementations to and because C is a language many programmers understand reasonably well [13]. `Ixy.hs` is a rewrite of the mentioned `ixy` user-space driver in Haskell. Instead of conciseness or lack of dependencies, its goals are to be idiomatic Haskell and to evaluate the suitability of Haskell as a language for writing drivers.

In the following chapters we look at how Haskell performed as a language for `ixy.hs`, what advantages and disadvantages it came with, and how it performed in various criteria.

Chapter 2

Ixy.hs

`Ixy.hs`, including forwarding application, consists of 1359 lines of Haskell code (including comments), and so is a bit larger than the original `ixy` implementation, that is around 1000 lines of C. An often heard statement is, that functional languages result in more compact code. In general this is true, but for code that is nearly entirely monadic, like `ixy.hs`, this statement doesn't hold up. Since lines of code are not a good metric of code quality we examine some more interesting implementation details of `ixy.hs`.

2.1 Architecture

In the following sections we examine how `ixy.hs` implements the architecture of the original `ixy` implementation in Haskell in detail.

2.1.1 Queues, Descriptors and DMA

An Intel 82599 NIC supports up to 128 receiving, and 128 transmitting queues to be active at the same time. The simplest case is the use of one receive and one transmit queue. A setup with multiple receive queues allows splitting up packets according to filters or hashes, while multiple transmit queues are merged in the NIC [7].
Each queue requires a block of memory it can store its descriptors in. For this purpose the driver requires a `hugetlbfs` to be mounted at `/mnt/huge`, where the driver allocates hugepages to be used as memory for the queues. In `ixy.hs` allocation of memory for queues is done in the `allocateMem` function defined in `Memory.hs`. The function is shown in its entirety in Listing 2.1.

It first opens a temporary file in the directory `/mnt/huge` by using `openBinaryTempFile`. Since `/mnt/huge` is a `hugetlbfs` filesystem the created file is a hugepage. A convenient

```haskell
allocateMem
:: (MonadThrow m, MonadIO m, MonadLogger m)
  => Int
  -> Bool
  -> m (Ptr a)
allocateMem size contiguous = do
  $(logDebug)
    $  "Allocating a memory chunk with size "
    <> show size
    <> "B (contiguous="
    <> show contiguous
    <> ")."
  let s = if size `mod` hugepageSize /= 0
        then shift (shiftR size hugepageBits + 1) hugepageBits
        else size
  liftIO $ do
    (_, h) <- PathIO.openBinaryTempFile (Path.absDir "/mnt/huge")
                                        (Path.relFile "ixy.huge")
    PathIO.hSetFileSize h $ fromIntegral s
    let f = memoryMap Nothing
                      (fromIntegral s)
                      [MemoryProtectionRead, MemoryProtectionWrite]
                      MemoryMapShared
    ptr <- bracket (handleToFd h) closeFd (\fd -> Just fd `f` 0)
    memoryLock ptr $ fromIntegral s
    return ptr
```

Listing 2.1: `allocateMem` defined in `Memory.hs`

```
data ReceiveDescriptor =
     ReceiveRead { rdBufPhysAddr :: {-# UNPACK #-} !Word64
                 , rdHeaderAddr :: {-# UNPACK #-} !Word64 }
     | ReceiveWriteback { rdStatus :: {-# UNPACK #-} !Word32
                        , rdLength :: {-# UNPACK #-} !Word16}
```

Listing 2.2: Representation of an advanced receive descriptor in `ixy.hs`

feature of using `openBinaryTempFile` is that it generates a unique random name for the temporary file. Next `allocateMem` `mmap`s the just allocated hugepage and `mlock`s it. A pointer to the beginning of the hugepage is then returned to the caller. This memory can then be used by the driver to store the descriptors of a queue. These descriptors serve as a channel of communication between the NIC and the driver. The NIC provides a choice between using legacy and advanced descriptors. `Ixy.hs` implements advanced descriptors, as does the original implementation. The representations of receive and transmit descriptors can be seen in Listings 2.2 and 2.3 respectively. Take note, that both descriptors are missing some fields, when compared to the C implementation. Unused fields were simply never implemented, but could be added by adding them in the types declaration and correctly adjusting the `Storable` implementation of the type. There are two noteworthy aspects about both these descriptor types. Firstly each type of descriptor has two layouts in memory, depending on who last wrote the descriptor. If the descriptor is written by the driver it needs to follow the layout specified by `ReceiveRead` or `TransmitRead`, if instead it is written by the NIC it follows the layout of `ReceiveWriteback` or `TransmitWriteback`. Essentially `ixy.hs` uses a sum type to represent a C `union` in this case. The second noteworthy aspect is the use of the `{-# UNPACK #-}` pragma. A strict field decorated with this pragma is unpacked by the compiler and directly inserted into the memory representation of the parent type to remove a level of indirection [1]. Additionally we implement the `Storable` typeclass for both descriptors. The implementation for `ReceiveDescriptor` is shown in Listing 2.4. An implementation for `TransmitDescriptor` follows analogously. Implementing this typeclass gives us access to three particularly convenient functions for descriptors: `peek`, `poke` and `sizeOf`. These functions, once implemented, take care of the layout of the `ReceiveDescriptor`, when it is marshalled to memory or demarshalled from memory.

2.1.2 Memory pools and Packet Buffers

In the previous section we explained how descriptors are stored and what their purpose is. In this section we have a look at another integral structure of operation called packet

```
data TransmitDescriptor =
     TransmitRead { tdBufPhysAddr :: {-# UNPACK #-} !Word64
                  , tdCmdTypeLen :: {-# UNPACK #-} !Word32
                  , tdOlInfoStatus :: {-# UNPACK #-} !Word32 }
     | TransmitWriteback { tdStatus :: {-# UNPACK #-} !Word32 }
```

Listing 2.3: Representation of an advanced transmit descriptor in `ixy.hs`

`ReceiveDescriptor`

```
instance Storable ReceiveDescriptor where
  sizeOf _ = 16
  alignment = sizeOf
  peek ptr = do
    status <- peekByteOff ptr 8
    len <- peekByteOff ptr 12
    return ReceiveWriteback {rdStatus=status, rdLength=len}
  poke ptr (ReceiveRead bufPhysAddr headerAddr) = do
    poke (castPtr ptr) bufPhysAddr
    pokeByteOff ptr 8 headerAddr
  poke _ (ReceiveWriteback _ _) = return $
       panic "Cannot poke a writeback descriptor."
```

Listing 2.4: Implementation of the `Storable` typeclass for `ReceiveDescriptor`.

```
data MemPool = MemPool { mpBaseAddr :: Ptr Word8
                       , mpNumEntries :: Int
                       , mpFreeBufs :: Array.IOUArray Int Int
                       , mpTop :: IORef Int
                       }
```

Listing 2.6: Implementation of a memory pool for packet buffers in `ixy.hs`

buffers. Packet buffers are essentially buffers for packets with some additional metadata storage.

```
data PacketBuf = PacketBuf { pbId :: Int
                           , pbAddr :: PhysAddr
                           , pbSize :: Int
                           , pbData :: ByteString }
```

Listing 2.5: Implementation of packet buffers in `ixy.hs`.

In Listing 2.5 we can see the implementation of a packet buffer in `ixy.hs`. A few interesting aspects about this implementation are the `newtype PhysAddr`, which is a wrapper around a `Word64` and the packet's data being stored as a `ByteString`. The `newtype` is a zero-cost abstraction, that disallows using a generic `Word64` where a `PhysAddr` is expected. The data being presented as a `ByteString` is a quite idiomatic approach in Haskell. Looking at both descriptor types again (Figures 2.2 and 2.3), we can see, that both have fields that point to a packet buffer. In the case of `ReceiveDescriptor` that field is `rdBufPhysAddr`, while for `TransmitDescriptor` it is `tdBufPhysAddr`. The NIC assumes that the packet buffer associated with descriptor is at the physical address specified by these fields. So we necessitate of a way to translate the virtual address of our packet buffers to their physical equivalents. In `ixy.hs` the `translate` function defined in `Memory.hs` fulfills this criterion. It is in essence the same as its equivalent in the original implementation, so we won't go into it in further detail.
Since packet buffers are reusable and the mapping between descriptors and them is arbitrary, they are organized by a data structure called memory pool. The pool provides a currently unused packet buffer to the user, when requested, and allows manually freeing the allocated buffer to throw it back into the pool. The definition for `ixy.hs`'s memory pools, called `MemPool`s, is shown in Listing 2.6. As we can see the implementation of a memory pool in `ixy.hs` is basically the same, as the one in the original implementation. The only noteworthy aspect of `MemPool` is, that it uses an `IOUArray`, which is a mutable unboxed array in the IO monad [4]. Being a mutable data structure it is

not particularly idiomatic to use `IOUArray`, but it is certainly the option with the least boilerplate. The `mpFreeBufs` array has both `Int` keys, as well as `Int` values. We make use of this by recording a mapping of descriptor to packet buffer in this array. Since both descriptors and packet buffers have a fixed size, we can use their position in the respectively allocated memory blocks as an integer id. E.g. descriptor number four can be found at $base + 4 * 16$, where base is the beginning of the descriptor memory block, and 16 is the size of the descriptor in bytes.

2.1.3 Mutability and IORefs

One property Haskell promises is immutability – the inability to change a expression once it has been evaluated. Generally immutability can result in being able to more easily reason about code. When working with an external device, that holds some state information, this inability to easily reflect changes in state is quite inconvenient. Haskell provides some ways around having to juggle immutable variables all throughout your code. The less interesting way is to use the `State` monad. This monad is similar to the `Reader` monad in function, but instead of allowing read-only access to the contained value, it also allows writing of the value. When the monad is applied to a function and the value is modified, a new immutable variable is implicitly returned. Essentially this is only syntactic sugar around the original approach of juggling immutable variables. The more interesting way is the use of `IORef`s (and `StRef`s). These types provide real mutability, as one would expect from a imperative language. An `IORef` holds a reference to a variable, that can be modified by functions like `modifyIORef` or `writeIORef`. In particular this allows us to have mutable fields within immutable data structures, which `ixy.hs` uses plenty. Listing 2.7 provides an example of exactly this scenario. It is required to be able to modify the index of the queue (which represents the queue's RDT) without the additional overhead of having to define a whole new queue, which in turn would prompt a redefining of the structure queues are stored in. With the use of `IORef` we avoided this huge overhead of having to create a new version of this huge nested data structure. Had we not done this a rewrite would be required after every single batch of packets that is received leading to an enormous deterioration in performance.

2.2 Implementation

In this section we examine `ixy.hs`'s API.

```
    indexRef <- liftIO $ newIORef (0 :: Int)
    return $! RxQueue
      { rxqDescriptor = descriptor
      , rxqMemPool    = memPool
      , rxqMap        = m
      , rxqIndexRef   = indexRef
      }
```

Listing 2.7: Excerpt from `Queue.hs` that shows use of `IORef`.

2.2.1 Initializing a device

A user of `ixy.hs` can initialize his Intel 82599 device by using the provided `newDriver` function. The signature of the `newDriver` function is seen in Listing 2.8.

```
newDriver
  :: (MonadCatch m, MonadThrow m, MonadIO m, MonadLogger m)
  => Text -- ^ The 'BusDeviceFunction' of the device.
  -> Int -- ^ The number of rx queues to initialize.
  -> Int -- ^ The number of tx queues to initialize.
  -> m (Maybe Device)
```

Listing 2.8: The signature of the `newDriver` function.

So to successfully initialize a device the user needs to provide a valid PCI identifier, and the number of receive queues, as well as transmit queues. Under the hood this function calls `Ixgbe.init`, which performs a substantial amount of initialization work, before any work can be done by the user.
The first step is to unbind any currently active drivers from the target device. This is done by writing the BDF of the target device to the `unbind` file of the active driver. Next we enable direct memory access for the device. This is required to let the driver and the device read/write from the same memory location, where we store descriptors and packets. After both unbinding and enabling DMA are done we can `mmap` the device's `resource` file, which allows us access to the base address registers (BAR) of the device. These allow us to configure the device to our liking, by writing and reading to specific offsets in the `mmap`-ed area. The mapping of offset to register can be taken from the Intel 82599 datasheet.
Now that we have access to the registers of the device we can begin the initialization as described in the Intel 82599 datasheet. An overview of the steps is provided here:

- Disable interrupts.

- Issue global reset and perform general configuration.
- Wait for EEPROM auto read completion.
- Wait for DMA initialization done.
- Setup the PHY and the link.
- Initialize all statistical counters.
- Initialize receive.
- Initialize transmit.
- Enable interrupts.

[11]
Additionally `ixy.hs` puts the initialized device into promiscous mode. This can be controlled by calling the provided `setPromisc` function. Further `ixy.hs` does not employ interrupts, since they are not beneficial for the high throughput rates we aim to achieve, and so does not re-enable interrupts, like the last step in the list suggests.

2.2.2 Reading and writing device registers

As in the original implementation the device's BAR is `mmap`-ed into the driver's working memory to allow read and write operations. To execute these operations we have a few helper functions in `Ixgbe.hs` that can read and write to the `mmap`-ed area. Of primary importance are the functions `get` and `set` that allow reading and writing of a device register respectively.

```
set :: Device -> Register -> Word32 -> IO ()
set dev register = pokeByteOff (devBasePtr dev) (fromEnum register)
```

The `set` function is essentially just a convenient wrapper around `pokeByteOff`. The latter allows writing a value to a pointer offset by some amount of bytes. The function extracts the base pointer to the `mmap`-ed BAR of the device, does a conversion from human-friendly register name to the register offset in the BAR, and writes the provided `Word32` value into that register.
The `get` function is similar to the aforementioned `set` function, with the exception of using `peekByteOff` in place of `pokeByteOff`. This allows it to read a value from the specified memory location, which in turn allows us to read a device register from the `mmap`-ed area.

```
get :: Device -> Register -> IO Word32
get dev register = peekByteOff (devBasePtr dev) $ fromEnum register
```

There exist a few other helper functions related to the registers in the code base, these being:

- `setMask` – Sets a bit mask in a register
- `clearMask` – Clears a bit mask in a register
- `waitSet` – Wait until all bits of the bit mask where set
- `clearMask` – Wait until all bits of the bit mask were cleared

2.2.3 Receiving packets

After the user has completed the initialization process described in 2.2.1, the device is ready to receive and send packets. Receiving is done by calling the `receive` function in `Ixgbe.hs`. Its signature is provided below:

```
receive :: Device -> Int -> Int -> IO [Ptr PacketBuf]
```

This signature requires a bit of explanation. The first argument is the `Device` we have initialized by calling the `newDriver` function in 2.2.1. The second argument is the identifier of the receive queue we want to poll. Queue ids are sequential increasing integers starting with the number 0. The third argument is the maximum amount of packets to receive in one go. This setting is called batch size, and can have a large effect on performance as we'll see in Chapter 3. Lastly the function returns a list of `Ptr PacketBuf`, where `PacketBuf` is a wrapper around a more user-friendly `ByteString` that contains the actual received packet.

2.2.4 Sending packets

Sending packets works much in the same fashion as receiving packets. The signatures for `receive` and `send` also have a certain similarity.

```
send :: Device -> Int -> MemPool -> [Ptr PacketBuf] -> IO ()
```

The first argument here is the `Device` we want to send packets from. The second argument is the identifier of the transmit queue that holds the packet, until it is sent. The third argument is the memory pool of the queue the packet was received from. This allows the `send` function to free up any buffers whose packets were already sent. For

convenience `Ixgbe.hs` has a helper function called `memPoolOf`, that allows extracting the memory pool of a receive queue.

2.2.5 Retrieving statistics

The NIC provides some statistics about the packets it handles in a few specific registers. `Ixy.hs` exposes only the amount of received/sent packets, and the amount of bytes received/sent. The user may call the `stats` function with the desired device as argument to retrieve the statistics.

```
stats :: Device -> IO Stats
```

Retrieving the statics resets them to zero as a side effect. The available fields of the `Stats` type correspond in order to: received packets, transmitted packets, received bytes, and transmitted bytes.

```
data Stats = Stats { stRxPkts :: Int
                   , stTxPkts :: Int
                   , stRxBytes :: Int
                   , stTxBytes :: Int }
```

Chapter 3

Evaluation

In this chapter we have a look at how `ixy.hs` performed under various circumstances.

3.1 Performance

All aspects tested in the following were performed with same measurement setup shown in Figure 3.1. The forwarding application is run on a server with a maximum CPU frequency of 3.3GHz (with turbo disabled) and takes control of two Intel 82599 NICs. It then endlessly forwards packets between the two NICs, while modifying one byte of each packet, to simulate a workload. Packets are generated with the use of MoonGen [5] on a second server with two Intel 82599 NICs as well. The packet generator expects to receive the packets it sent out on one of its NICs on its respective equal. To test the full range of bandwidth a Intel 82599 NIC can provide we send at full load on both NICs, which means a combined packet rate of 29.76Mpps at a packet size of 64 byte Special attention was payed to using two single-ported NICs on the forwarder site, to avoid any issues that may arise from missing PCIe bandwidth.

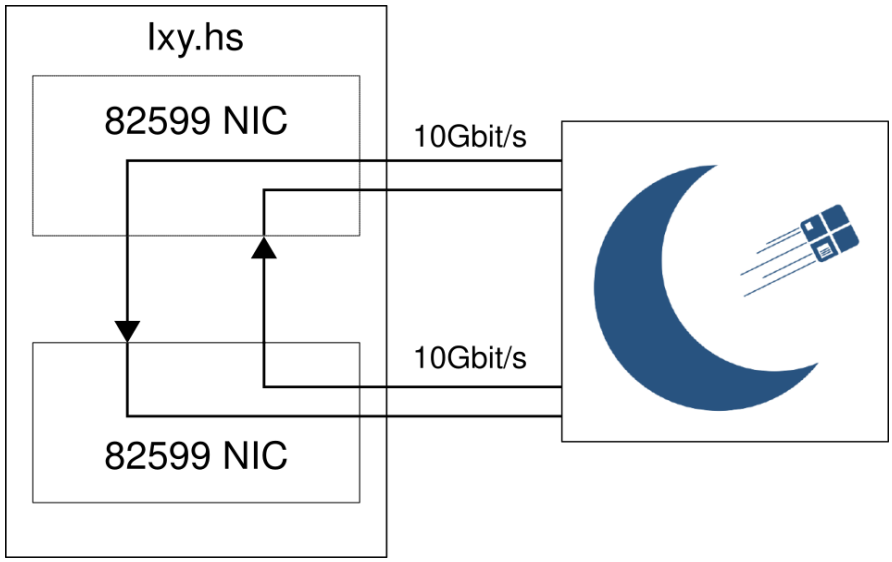

Figure 3.1: An illustration of the setup used to benchmark `ixy.hs`.

3.1.1 Comparison of Backends

`Ixy.hs` uses the Glasgow Haskell Compiler[1] at version 8.4.3 to generate its binaries. With GHC, as of now, there are two available code generation backends:

- GHC's native backend
- GHC's LLVM backend

These backends can be enabled by passing either `-fasm` or `-fllvm` to the compiler respectively. The difference between these backends is in the code they generate. The native backend is the default option for GHC. It instructs the compiler to compile the Haskell code to an intermediate language called GHC Core. GHC Core is a explicitly-typed modified version of a lambda calculus, that the compiler then uses to generate assembly instructions for the platform it currently runs on. When the LLVM backend is enabled GHC compiles the before mentioned GHC Core into LLVM IR, an intermediate language used by the LLVM compiler to generate assembly for the current platform. As we'll see, in some cases using one or the other backend may result in better performance.

[1] `https://www.haskell.org/ghc/`

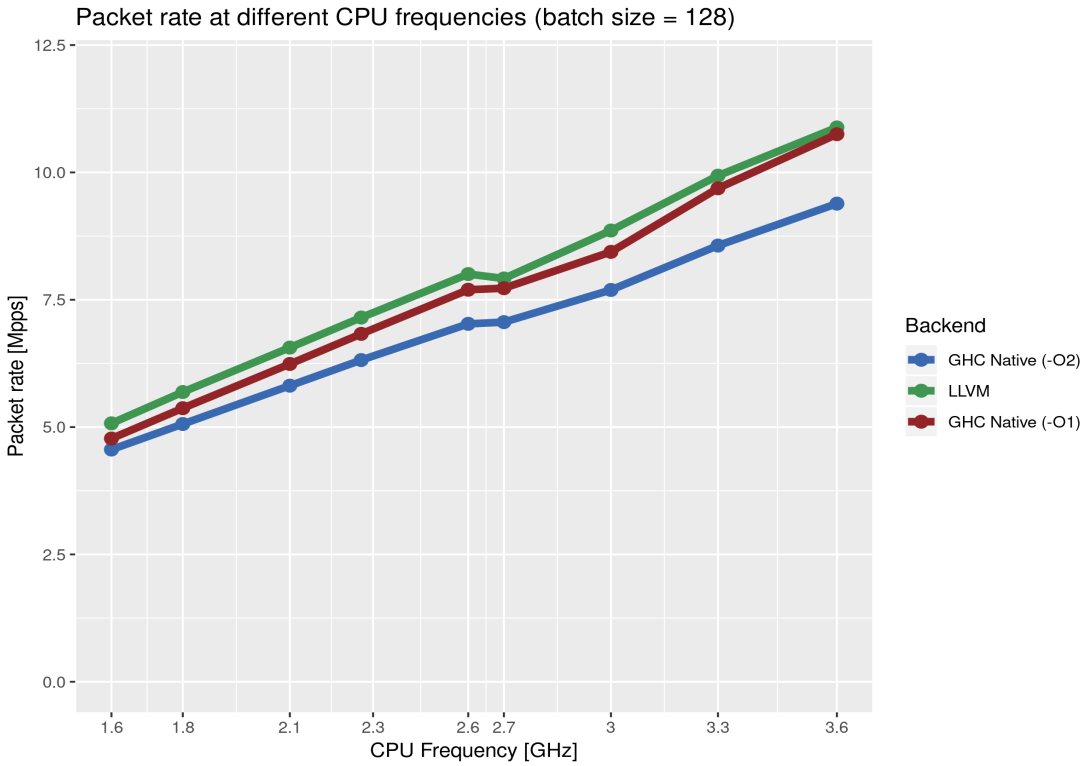

Figure 3.2: Packet rate vs. CPU frequency for different backends and configurations.

To test the performance differences between these backends, we performed measurements of the average throughput of packets at different CPU frequencies using the before mentioned test setup. Figure 3.2 shows the results of these measurements. We also tested the native backend twice with differing optimization options, since less optimization seemed to give better results than more optimization for an unknown reason. For the test we also chose a batch size of 128 packets, since this seemed to give the best overall results. As can be seen in Figure 3.2, all configurations start out close together at around 4.9Mpps. There is a notable approximately linear increase in performance with increasing frequency for all configurations. Noteworthy is that the jump from 2.6GHz to 2.7GHz seems to be at least problematic for all configurations, with the native backend's performance rising at a way slower pace then before, and the LLVM backend's performance even deteriorating. There is a visible constant trend of inferiority for the more optimized version of the native backend, making it the clear loser of this comparison. It maxes out at around 9.4Mpps, while the other two configurations show very similar performance and top out at around 10.1Mpps. The LLVM backend is ahead at every sampling point though, making it the overall winner in terms of throughput performance.

Frequency [GHz]	Native (-O2) [%]	Native (-O1) [%]	LLVM [%]
1.6	–	+4.7	+11.3
1.8	–	+6.1	+12.4
2.1	–	+7.3	+12.9
2.3	–	+8.1	+13.2
2.6	–	+9.5	+13.9
2.7	–	+9.4	+12.1
3.0	–	+9.7	+15.2
3.3	–	+13.1	+16.0
3.6	–	+14.5	+15.9

Table 3.1: Percentual difference in performance of configurations based on worst-performing configuration for the benchmark shown in Figure 3.2.

3.1.2 Batching

Another interesting criteria that can have an impact on performance is the batch size (see 2.2.3) the driver uses to receive and send packets. For every batch of packets received or sent the driver must write to the queue registers on the NIC through the PCIe bus. At very high throughput rates this can prove to be a bottleneck, which means minimizing the amount of writes by receiving and sending in larger batches should have a positive effect in performance. To test this we once again make use of the setup mentioned in the beginning of this chapter. We compare the three backend configurations explained in 3.1.1, by varying their batch sizes at a constant CPU frequency.

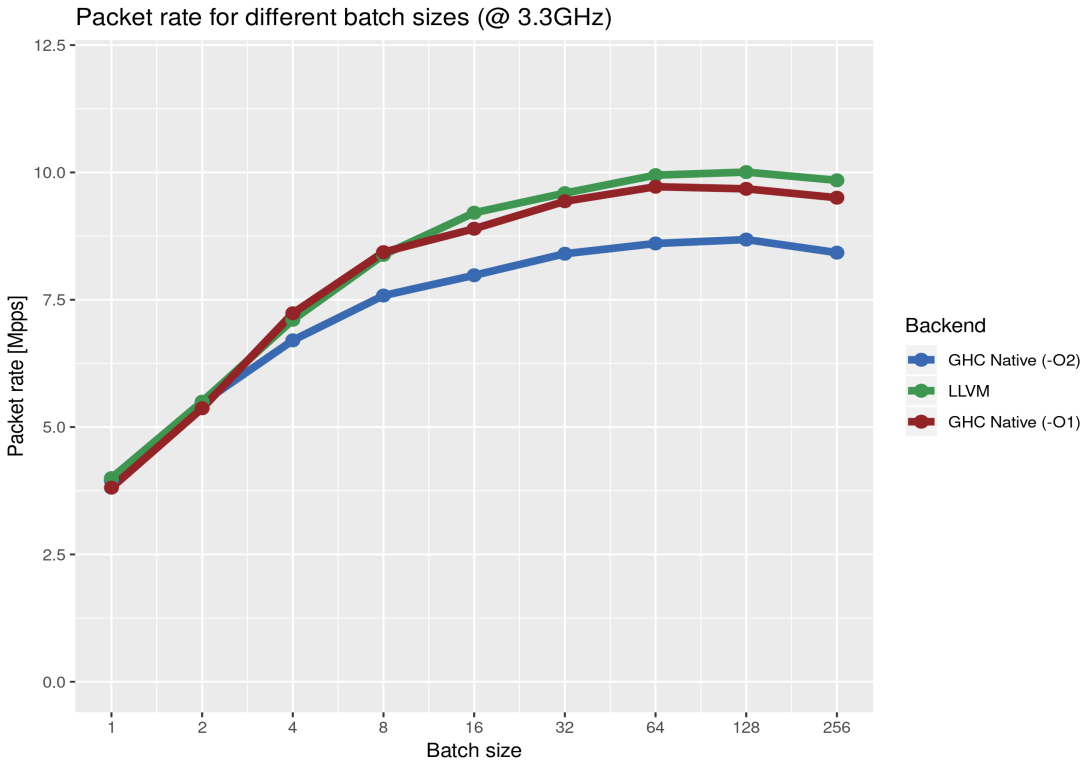

Figure 3.3: Packet rate for different batch sizes at 3.3 GHz CPU frequency.

Figure 3.3 shows the results of this test for a CPU frequency of 3.3GHz (the maximum frequency supported by the CPU, without turbo). It is immediately visible that batch sizes does have an impact on performance, as suspected. At batch size one all configurations are very close to each other in terms of throughput, at around 3.9Mpps, suggesting that the batch size is actually an extremely large limiting factor at this point. The same is true for a batch size of two, where the configurations are still lumped together on the plot at around 5.4Mpps. Interestingly the less optimized native configuration performs slightly better than the LLVM configuration at batch sizes four and eight. The optimized native and the LLVM configuration reach their peak performance at a batch size of 128, after which performance deteriorates again. For the less optimized native configuration the peak can be found at a batch size of 64. The LLVM and the less optimized native configuration had very similar peak performances of approximately 10Mpps and 9.7Mpps respectively, while the optimized native configuration could only achieve a peak throughput rate of 8.7Mpps. This means the optimized native configuration performed approximately 13% worse, than the LLVM configuration. A last interesting aspect of this test is the deterioration in performance after the respective peak batch sizes of 64 and 128. This may be a result of increasing cache misses with further increasing batch

Batch size	Native (-O2) [%]	Native (-O1) [%]	LLVM [%]
1	+3.5	–	+4.9
2	+2.0	–	+2.4
4	–	+7.9	+6.0
8	–	+11.2	+10.5
16	–	+11.5	+15.4
32	–	+12.3	+14.2
64	–	+13.0	+15.6
128	–	+11.5	+15.3
256	–	+12.8	+16.9

Table 3.2: Percentual difference in performance of configurations based on worst-performing configuration for the benchmark shown in Figure 3.3.

sizes.

At a CPU frequency of 3.3GHz there is a remarkable difference between the various tested configurations at different batch sizes. For these next measurements we adjust the CPU frequency to 1.6GHz (the minimum supported frequency) and once again perform measurements with increasing batch sizes.

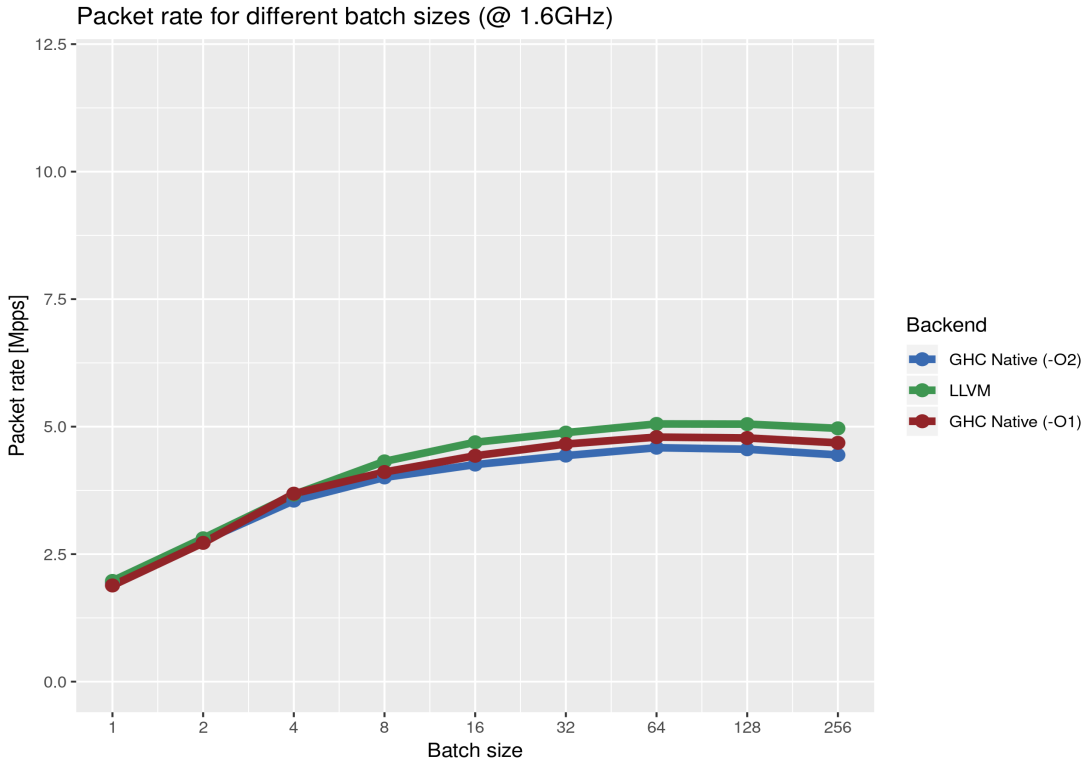

Figure 3.4: Packet rate for different batch sizes at 1.6 GHz CPU frequency.

Batch size	Native (-O2) [%]	Native (-O1) [%]	LLVM [%]
1	+2.7	–	+4.7
2	+1.4	–	+3.3
4	–	+3.8	+3.3
8	–	+2.6	+7.9
16	–	+4.1	+8.2
32	–	+5.1	+10.2
64	–	+4.5	+10.2
128	–	+4.9	+10.8
256	–	+5.3	+11.7

Table 3.3: Percentual difference in performance of configurations based on worst-performing configuration for the benchmark shown in Figure 3.4.

The results are shown in Figure 3.4. Unsurprisingly the performance of each configuration has dropped quite sharply, compared to the previous measurements (see Figure 3.3). The LLVM configuration once again outperforms the other configurations, but this time reaches its peak performance of approximately 5.0Mpps at a batch size of 64, instead of 128. In fact this is true for all configurations. The optimized native configuration is again the worst in throughput performance, but this time is only approximately 10% worse compared to the LLVM configuration's performance.

3.1.3 Garbage Collection

When writing performance-critical software languages with garbage collection are often immediately disqualified, based on the assumption that garbage collection pauses will prove to be a large detriment to the performance of the application. The next benchmark looks at the impact garbage collection has on `ixy.hs`. Figure 3.5 holds the results of the benchmark. Surprisingly the optimized native configuration wins out over the other two configurations this time, by spending only a maximum of 0.6% of execution time in garbage collection. The less optimized native configuration and the LLVM configuration reach their peak values of 1.4% and 1.6% of execution time respectively at a batch size of 256. Figure 3.5 also shows a slight increase in time spent in garbage collection with increasing batch size. This is more noticeable in the worse-performing configurations. The optimization level of the optimized native configuration may have had a large impact on the garbage collection and that's why this configuration performs so well in this test. Combining these results with the previous benchmarks leads to the conclusion, that garbage collection does not have a significant performance impact in `ixy.hs`. This conclusion comes about by making a few observations. The first would be, that configurations that perform worse in the garbage collection benchmark, perform significantly better in the other benchmarks. Secondly time spent in garbage collection

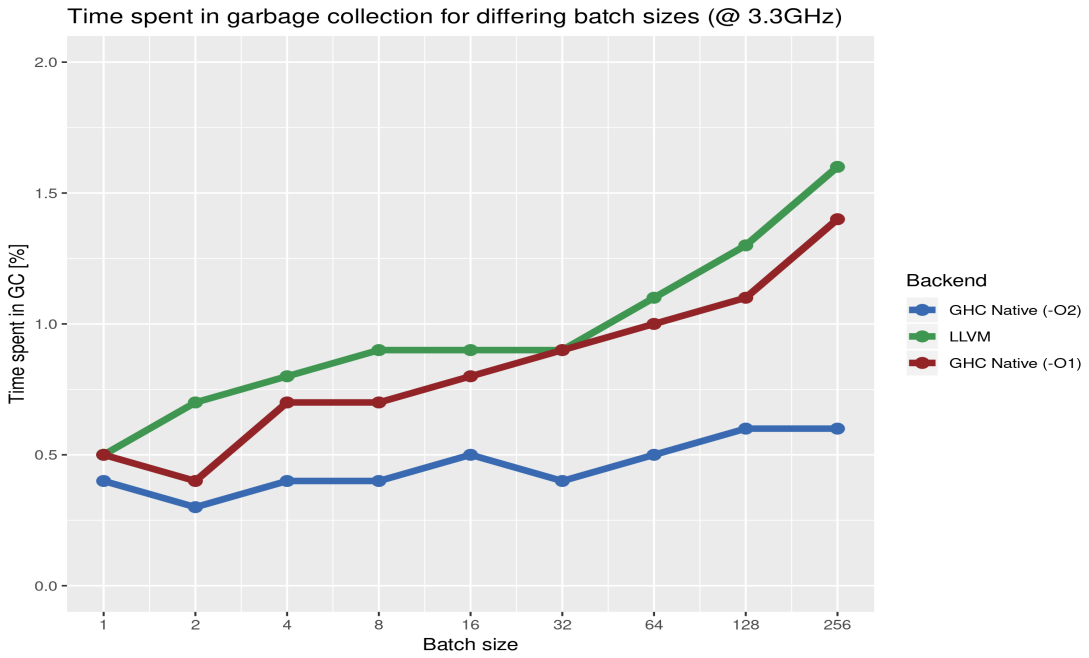

Figure 3.5: Time spent in GC for different backends and configurations.

is quite little. Even the worst-performing configuration only spends a maximum of 1.6% of execution time in garbage collection, which at the level `ixy.hs` can operate at, would be at most an insignificant increase in throughput of approximately 160000 packets, if that time could be eliminated entirely.

Enabling the threaded GHC runtime with the compiler argument `-threaded` also had an adverse effect on both throughput rate and garbage collector pause time, reducing throughput by approximately 8% and raising garbage collector times into the millisecond range, leading to dropped packets.

3.1.4 Comparison to other languages

Finally let's compare `ixy.hs`'s performance to implementations in different languages. These implementations all follow the same architecture as the original C implementation does, although with their own language-specific quirks. Whenever possible, these implementations try to use language-specific features, that are not usually seen in drivers, since they are mostly written in C. Implementations can be found in [14].

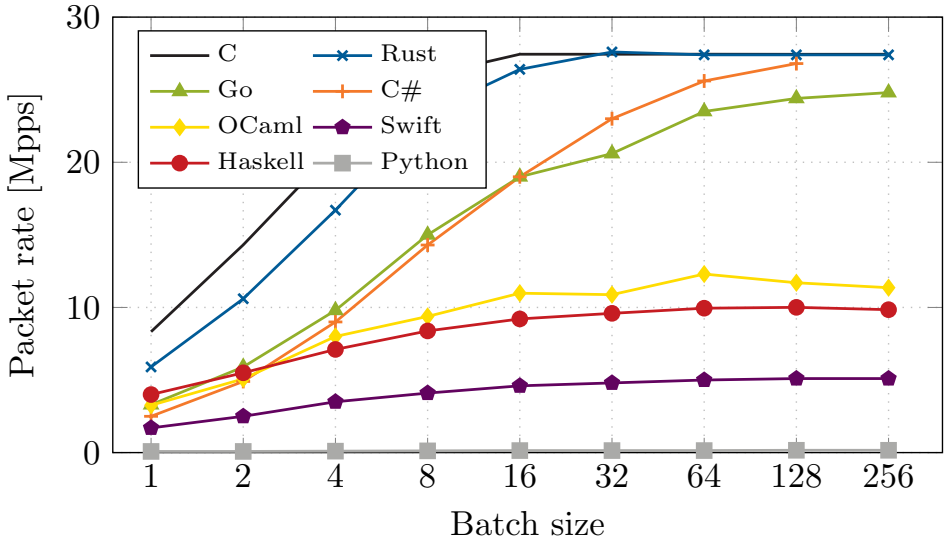

Figure 3.6: Packet rate vs. batch size (@3.3GHz) for different languages [6].

Figure 3.6 shows a comparison of packet rate for different batch sizes, like we saw in 3.1.2, but for all languages that have an implementation of `ixy`. As we can see `ixy.hs` performs rather badly overall. Languages that have an affinity for systems programming, like C, Rust and Go, (and C#) perform satisfyingly well. C outperforms Haskell for every batch size. Even at the minimum batch size of 1, C is 108% faster than Haskell. This speed difference increases to a maximum of 174% at a batch size of 128. The story is very similar for Rust, Go and C#. An interesting observation is that OCaml, Haskell's only functional brother in the imperative-dominated world of `ixy`-implementations, shows quite similar performance to Haskell. Although OCaml mostly outperforms Haskell by a small margin, in the grand scheme of things these two functional languages achieve around the same performance. This may be, because functional languages mostly are not particularly concerned about achieving the highest possible performance.

There is only one dynamically-typed language in the pool of implementation languages: Python. With the sample size being so small making a conclusion about dynamically-typed languages' performances would be unsupported, but dynamic typing should be disadvantage for the development of drivers in general. Dynamic typing increases the cognitive load of using a variable correctly for the programmer. Another interesting observation is the division between weakly and strongly typed languages. The above figure doesn't show any evidence of strongly typed languages being more performant than weakly typed languages. An example for this can be seen by looking at C and

Rust. Both languages perform similarly well, but the former is weakly-typed, while the latter is strongly-typed. A strong type system, like Rust's or Haskell's, has the programmer dealing with less unintended behavior, which once again is an advantage in the development of drivers.

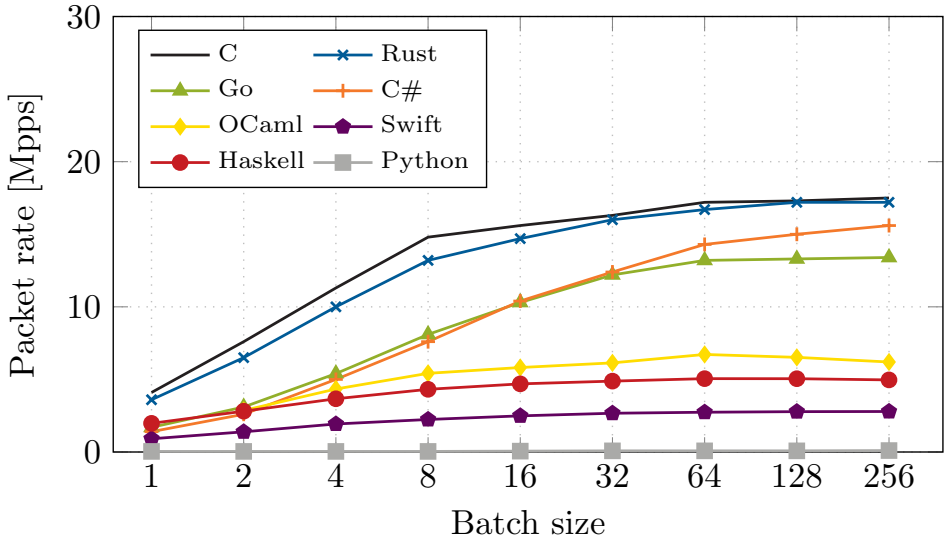

Figure 3.7: Packet rate vs. batch size (@1.6GHz) for different languages [6].

In Figure 3.7 we can once again see the packet rate for different batch sizes for all languages present in Figure 3.6, but this time the measurements were made at a CPU frequency of 1.6GHz. As could already be gathered in section 3.1.2 lowering the CPU frequency has a negative impact on the throughput rate of `ixy.hs` at all batch sizes. As expected the same is true for all other implementations of `ixy` too. Changing the CPU frequency had no impact on the ranking of the implementations, when compared with Figure 3.6.

3.2 Profiling

In this section we compare how much time was spent in specific locations of the code of `ixy.hs` compared to the original `ixy` implementation and finally look at `ixy.hs`'s memory behavior.

Cost Centre	Module	%Time	%Alloc
receive.go	Lib.Ixgbe	35.9	22.2
send.go	Lib.Ixgbe	16.4	4.7
rxGetMapping	Lib.Ixgbe.Queue	5.0	10.5
forward	Lib.Ixgbe	4.7	3.7
rxMap	Lib.Ixgbe.Queue	4.3	10.5
send.clean.cleanDescriptor	Lib.Ixgbe	4.2	5.8
allocateBuf	Lib.Memory	3.8	7.0
mkTxQueue.descriptor	Lib.Ixgbe.Queue	3.1	2.4
txGetMapping	Lib.Ixgbe.Queue	2.7	7.0
idToPtr	Lib.Memory	2.4	4.7
freeBuf	Lib.Memory	2.3	2.3
txMap	Lib.Ixgbe.Queue	2.3	10.4
receive.go.next	Lib.Ixgbe	1.4	2.3
rxqDescriptor	Lib.Ixgbe.Queue	1.3	0.0
txqDescriptor	Lib.Ixgbe.Queue	1.2	0.0
send.clean	Lib.Ixgbe	1.0	0.9
mkRxQueue.descriptor	Lib.Ixgbe.Queue	0.9	2.3
send.go.indexRef	Lib.Ixgbe	0.3	2.3

Table 3.5: More detailed profiling statistics for `ixy.hs`.

	RX	TX	Forwarding
`ixy`	44.8	14.7	12.3
`ixy.hs`	58.8	35.2	4.6

Table 3.4: Percent of execution time spent for different actions compared for `ixy` [7] and `ixy.hs`.

`ixy.hs` spends a larger percentage of execution time in both receiving and sending packets, but a smaller percentage for forwarding, which includes e.g., updating counters and printing statistics. Looking at Table 3.5 we can see that receiving and sending packets are by far the most time-consuming operations in `ixy.hs`. Another category of functions that appears rather time-consuming are `rxGetMapping` and `rxMap`. These functions are used to update the map that stores information about which packet buffer is mapped to which descriptor. The frequent use of these functions, and the modification of the underlying `IOUArray` they do, very much contribute to their time consumption.

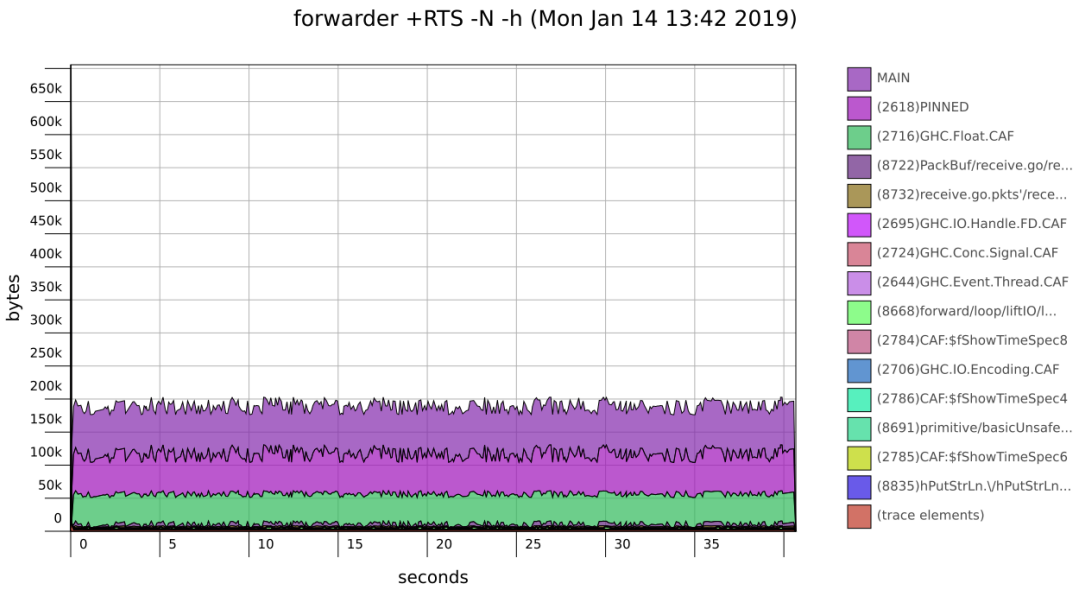

Figure 3.8: Plot of heap memory usage over time for `ixy.hs`.

As we can see from Figure 3.8, heap size never exceeds 205KB and remains relatively constant throughout the execution of the driver. This is to be expected, since in the benchmark scenario the driver is working at full load all the time. A further conclusion that can be drawn from the figure is that the driver does not have any space leaks, since these would be visible in the plot.

Chapter 4

Related Work

When compared to other programming languages, that explicitly name 'systems programming' as one of their strengths, the amount of systems software written in Haskell is minuscule. In the following we look at two projects that have used Haskell in at least some of their efforts.

4.0.1 House - A Haskell OS

House is an operating system written in Haskell. Its purpose is to enable exploring low-level and system programming on a system that explicitly supports high-level functional languages. It provides a monadic interface to memory management, hardware, user-mode processes, and low-level device I/O. By using Haskell House enforces memory safety in nearly all circumstances, in part due to formal assertions, that are written in a specially developed programming logic, called P-Logic [9]. The project implemented a separation kernel, a windowing system, device drivers including for PS/2, VBE2.0, NE2000 NICs, Intel PRO/100 NICs, and a network stack with support for Ethernet, IPv4, ARP, DHCP, ICMP, UDP, TFTP, and TCP [9].

Although the authors say their experience with development of House in Haskell was largely positive, in part due to the strong type system Haskell brings and the speed of compiled code, they also state, that they have not yet determined, whether devices requiring high bandwidth and low latency can be adequately serviced in a Haskell runtime [9].

4.0.2 PFQ

PFQ is a packet capturing engine for Linux that is highly optimized for multi-core architectures, or network devices with multiple hardware queues. It also includes a pure functional DSL, called *pfq-lang*, that is designed for the early stages of in-kernel packet processing [17]. pfq-lang is inspired by Haskell and its purpose is to be used as a language to build applications on top of network drivers. With the help of pfq-lang one can implement efficient applications such as forwarders, fire walls, or load balancers. The framework also includes user-space libraries for C,C++,Haskell and implementations of pfq-lang as a eDSL for C++ and Haskell [17].

PFQ uses a functional engine in its packet steering block [2], which determines how packets are to be processed. The possible options are: dropping the packet, returning it back to the kernel, forwarding it to specific socket, a random socket, or a group of sockets. All of these actions can be chosen according to different aspects of the incoming packet and are configured using pfq-lang. By making pfq-lang a functional language one can run verification checks against programs written in it, verify properties like the absence of loops or type correctness [18]. Having these checks helps against crashes that threaten system stability, since PFQ runs in kernel-space. Unlike `ixy.hs` PFQ is not a user-space driver framework. It relies on a kernel module written in C to do the heavy lifting of fetching packets from the kernel. In fact 75% of PFQ's code is written in C, with the rest being 10% C++, 8% Haskell and 7% various other languages (Makefile, HTML, M4, ...). The Haskell part of PFQ is the implementation of pfq-lang and the user library for PFQ.

Chapter 5

Conclusion

At this point we can answer the initial question: should one write network drivers in Haskell?

Without adding more context to the question there isn't a clear cut answer to be found. In scenarios that require very high throughput rates, like our test scenario outlined in Chapter 3, Haskell performs worse than other languages by a large margin. Using Haskell to write a driver that has to perform in these kinds of scenarios will likely result in missed performance goals. In scenarios, where very high throughput rates are not a requirement though, Haskell may just be a convenient choice of language. Haskell's strong type system eliminates a whole class of errors that some other languages bring with them, and aids the programmer in thinking more closely about the interfaces they write. Haskell's garbage collection turned out to have only a minor influence on performance, while providing the programmer with the convenience of not having to worry about memory management.

Overall Haskell is a language that is focused on helping the programmer write reliable software, which is a great property for device drivers. On the performance front Haskell underachieves compared to other languages like C, Rust or Go.

Chapter A

Appendix

Frequency [GHz]	Native (-O2) [Mpps]	Native (-O1) [Mpps]	LLVM [Mpps]
1.6	4.55924	4.77540	5.07358
1.8	5.06042	5.37116	5.68810
2.1	5.81322	6.23922	6.56096
2.3	6.31546	6.82990	7.15066
2.6	7.02694	7.69698	8.00436
2.7	7.0608	7.72674	7.91780
3.0	7.69232	8.44018	8.85952
3.3	8.56386	9.68864	9.93576
3.6	9.3874	10.75056	10.88282

Table A.1: Exact numbers for packet rate vs. CPU frequency plot shown in Figure 3.2.

Batch size	Native (-O2) [Mpps]	Native (-O1) [Mpps]	LLVM [Mpps]
1	3.94540	3.81090	3.99850
2	5.47472	5.36960	5.49784
4	6.70302	7.23488	7.10696
8	7.58266	8.43464	8.38162
16	7.98028	8.89450	9.20834
32	8.40360	9.43344	9.59380
64	8.60496	9.71902	9.94706
128	8.68164	9.67776	10.00632
256	8.42494	9.50376	9.84480

Table A.2: Exact numbers for packet rates at different batch sizes (@3.3GHz) plot shown in Figure 3.3.

Batch size	Native (-O2) [Mpps]	Native (-O1) [Mpps]	LLVM [Mpps]
1	1.937254	1.885668	1.974526
2	2.758160	2.721020	2.810620
4	3.549540	3.685580	3.667280
8	4.004200	4.110080	4.318720
16	4.256300	4.428840	4.693600
32	4.432480	4.658940	4.882680
64	4.586780	4.795000	5.053320
128	4.556800	4.777820	5.049420
256	4.445180	4.682200	4.966760

Table A.3: Exact numbers for packet rates at different batch sizes (@1.6GHz) plot shown in Figure 3.4.

Batch size	Native (-O2) GC Time [%]	Native (-O1) GC Time [%]	LLVM GC Time [%]
1	0.4	0.5	0.5
2	0.3	0.4	0.7
4	0.4	0.7	0.8
8	0.4	0.7	0.9
16	0.5	0.8	0.9
32	0.4	0.9	0.9
64	0.5	1.0	1.1
128	0.6	1.1	1.3
256	0.6	1.4	1.6

Table A.4: Exact numbers for time spent in garbage collection for differing batch sizes (@3.3GHz) plot shown in Figure 3.5.

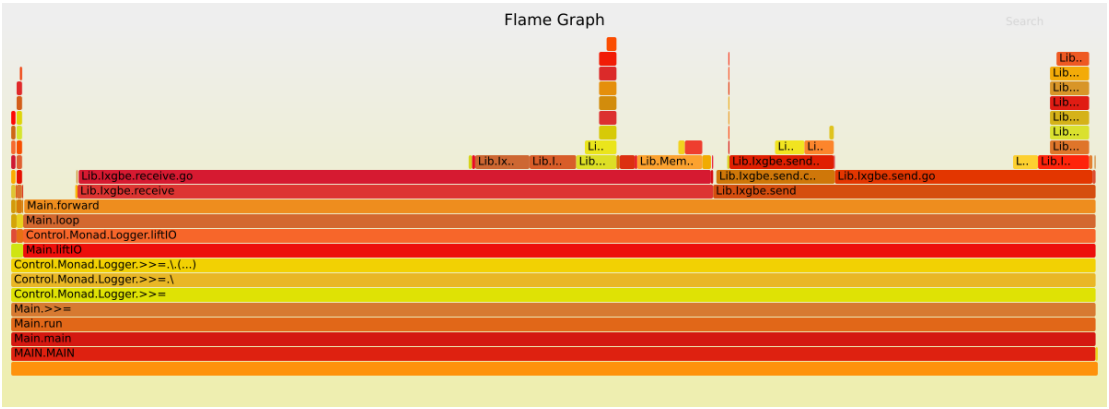

FIGURE A.1: Flamegraph for `ixy.hs` running in the LLVM configuration.

Chapter B

List of acronyms

ARP	Address Resolution Protocol.
BAR	Base Address Registers.
BDF	Bus Device Function Identifier.
DHCP	Dynamic Host Configuration Protocol.
DSL	Domain-specific language.
eDSL	Embedded domain-specific language.
EEPROM	Electronically Erasable Programmable Read Only Memory.
GC	Garbage Collection.
GHC	Glasgow Haskell Compiler.
ICMP	Internet Control Message Protocol.
LLVM	Low Level Virtual Machine.
NIC	Network Interface Card.
PS/2	IBM Personal System/2 Port.
TCP	Transmission control protocol. Stream-oriented, reliable, transport layer protocol.
TFTP	Dynamic Host Configuration Protocol.
UDP	User datagram protocol. Datagram-oriented, unreliable transport layer protocol.
VBE2.0	Vesa BIOS Extension 2.0.

Bibliography

[1] *7.1.3 Pragmas*. https://downloads.haskell.org/~ghc/7.0.3/docs/html/users_guide/pragmas.html. Accessed: 2018-01-17.

[2] Nicola Bonelli et al. "On Multi–gigabit Packet Capturing with Multi–core Commodity Hardware". In: *Passive and Active Measurement*. Ed. by Nina Taft and Fabio Ricciato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 64–73. ISBN: 978-3-642-28537-0.

[3] *Data Plane Development Kit*. https://www.dpdk.org/. Accessed: 2018-01-20.

[4] *Data.Array.IO*. http://hackage.haskell.org/package/array-0.5.3.0/docs/Data-Array-IO.html. Accessed: 2018-01-17.

[5] Paul Emmerich et al. "Moongen: A scriptable high-speed packet generator". In: *Proceedings of the 2015 Internet Measurement Conference*. ACM. 2015, pp. 275–287.

[6] Paul Emmerich et al. *The Case for Writing Network Drivers in High-Level Programming Languages*. 2019.

[7] Paul Emmerich et al. *Writing User Space Network Drivers*. 2019. eprint: arXiv:1901.10664.

[8] *GHC/Memory Management*. https://wiki.haskell.org/GHC/Memory_Management.

[9] Thomas Hallgren et al. "A Principled Approach to Operating System Construction in Haskell". In: *SIGPLAN Not.* 40.9 (Sept. 2005), pp. 116–128. ISSN: 0362-1340. DOI: 10.1145/1090189.1086380. URL: http://doi.acm.org/10.1145/1090189.1086380.

[10] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (1989), pp. 98–107. DOI: 10.1093/comjnl/32.2.98. eprint: /oup/backfile/content_public/journal/comjnl/32/2/10.1093/comjnl/32.2.98/2/320098.pdf. URL: http://dx.doi.org/10.1093/comjnl/32.2.98.

[11] *Intel 82599 10 GbE Controller Datasheet*. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf. Accessed: 2018-12-23.

[12] *Introduction - HaskellWiki*. https://wiki.haskell.org/Introduction. Accessed: 2018-12-15.

[13] *ixy - a userspace network driver in 1000 lines of code*. https://github.com/emmericp/ixy. Accessed: 2018-01-21.

[14] *Ixy Languages - The same high-speed user space network driver written in lots of different high-level languages.* https://github.com/ixy-languages/ixy-languages.

[15] *Lazy Evaluation - HaskellWiki*. https://wiki.haskell.org/Lazy_evaluation. Accessed: 2018-12-23.

[16] *Linus Torvalds on C++*. http://harmful.cat-v.org/software/c++/linus. Accessed: 2018-01-20.

[17] *PFQ - Functional Network Framework for Multicore Architectures*. https://github.com/pfq/PFQ. Accessed: 2019-01-30.

[18] Dominik Schöffmann. "Comparing PFQ: A High-Speed Packet IO Framework". In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 61 (2016).

[19] *Snabb: Simple and fast packet networking*. https://github.com/snabbco/snabb. Accessed: 2018-01-20.

[20] *Why Haskell matters - Haskell Wiki*. https://wiki.haskell.org/Why_Haskell_matters. Accessed: 2019-01-23.