Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS: GAMES ENGINEERING

Writing Network Drivers in Javascript

Sebastian Di Luzio

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

Writing Network Drivers in Javascript

Netzwerktreiber in Javascript schreiben

Author:	Sebastian Di Luzio
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich M. Sc.
Date:	July 15, 2019

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, July 15, 2019

Location, Date

Signature

Abstract

Writing drivers has been and still is considered a very unpleasant task. Some reasons for this are the circumstances such as the need to work in kernel space or having to rely on C. These times are over now, though, as kernel modules no longer have to be written in C and drivers no longer have to be written in the kernel space.

These user space drivers can be written using virtually any language but are still mostly created with C. The goal of ixy and this thesis is to show that user space network drivers can be written in other languages, and to analyze if they should. Another goal is to create readable and accessible implementations of network drivers for educational purposes.

We show that it is possible to write a state-of-the-art user space network driver with about 900 lines of code using idiomatic JavaScript and some C to help out, which is made possible by the Node.js runtime. Whilst not beating the C implementations performance, we do indeed beat our own expectations by forwarding around five million packets per second (4.95 mpps) on one single 3.3 GHz CPU core. This equals 18% of the C, 103% of Swift, and 3538% of Python implementations' performance.

We discuss the optimization the code went through and evaluate the implementation. Based on this information, we conclude if JavaScript can be considered a good programming language for writing network drivers.

Contents

List of Figures

Chapter 1

Introduction

Over the last few years, ever more user space drivers have come up. The main reason for this is performance. If drivers are written in kernel space, they have some unique priviliges, but they also need to constantly switch between kernel and user mode. This has an impact on performance that cannot be neglected, and therefore, more and more drivers used in production have been written in user space.

User space drivers do not only address this performance problem but also provide much easier development because kernel modules can only be written in a subset of the C programming language. Developing in user space adds the possibility to use debugging and other tools provided for the language of choice. Another downside is that C and subsets thereof are inherently unsafe. Poorly written or untested code can cause segmentation faults and potentially write over other programs, files and even crash the entire system. Using newer and safer languages can completely erase the possibilities of such errors.

Out of the need to bring user space drivers to the masses as well as explain and show well written and easily readable drivers, the Ixy-languages project (ixy) was created. Ixy is an educational user space network driver showcasing how to write such a driver. It is currently written in C as the lowest common denominator of programming languages [14] and focuses on the Intel 82599 Network Interface Card (NIC) [21]. There are already multiple implementations in other languages such as Rust, Haskell, Swift, C#, Go, OCaml and Python [13].

The goal of this thesis is to determine if JavaScript (JS) is also a suitable choice for user space drivers. We will write a minimal, yet fully functional driver in JS, and compare it to the reference C implementation as well as the other ixy implementations.

Chapter 2

Javascript as a language

There are many different programming languages for many different use cases [23]. We commonly distinguish between high- and low-level programming languages, where low-level means closer to the hardware and therefore potentially more suitable for controlling hardware such as a NIC via a driver. JS is a high-level, interpreted programming language that is mainly used within web applications and is widely spread as the standard web language along with HTML and CSS. This of course raises the question why we would use such a high-level programming language to develop a NIC driver.

According to this year's Stack Overflow developer survey [8], JS is the most used programming language and has been so for the last six years. About 70% of professional programmers know and use JS. For the goal of this project, which is bringing driver development closer to programmers' attention and showing the possibilities of user-space drivers, JS addresses the largest target group possible.

2.1 Syntax

As can be taken from Listing 2.1 below, the JS syntax is very similar to other widely used languages such as C#, Java and C. It uses curly brackets to define scopes for loops, functions, etc. Classes and similar structures exist in JS's prototype-based object-orientation as well. These familiarities make JS even more accessible and valuable for ixy, because it is easy to read even for programmers that are not used to developing with it. The biggest difference to most other languages is that typing is dynamic, so the syntax does not require nor support type declarations.

```
1 const smallNumber = 9; // JS knows this is a number
2 for (let i = 0 ; i < smallNumber ; i++ ) {
3   console.log("This will show up nine times.");
4 }
```

Listing 2.1: Simple example of JS syntax

Besides dynamic typing, another possible source of confusion can be the fact that syntax such as semicolons are optional by default and e.g. strings can use different types of quotes. In order to make the syntax more consistent, we included ESLint, which is a static code analysis tool used in software development for checking if JS source code complies with coding rules. We added the rules used by Airbnb [2] with some of our own adjustments. This ensures that the code style is consistent and follows well established development and security guidelines.

```
1 const someString = 'this is some string' // semicolons are optional
2 const someOtherString = "this is some string"
3 const anotherString = `this is some string`;
4 console.log(someString == someOtherString == anotherString); // true
```

Listing 2.2: Example of confusing JS which is not possible with our ESLint setup

2.2 Typing

Development in JS is very quick and easy because it uses dynamic typing. Especially in larger projects, this can lead to unstable and badly documented code where runtime errors could happen at any moment. But for small- to mid-scale projects, these issues are not as relevant since most parameters do not get passed through multiple levels of functions and abstractions and therefore, have implicit types and classes.

```
1 let smallNumber = 9; // JS knows this is a number and not constant
2 const smallNumberAsString = '9'; // JS knows this is a string
3 const objectWithNumber = { num: 12 }; // JS knows this is an object
4 objectWithNumber.str = 'hello'; // we can change attributes easily
5 smallNumber = objectWithNumber; // we can easily change its type
6 console.log(smallNumber); // object : { num : 12, str : 'hello }
```

Listing 2.3: Example of useful but potentially confusing dynamic typing

There is a solution for these problems which is called TypeScript. It is a strict syntactical superset of JS and adds optional static typing to the language, which essentially makes development safer and more structured, but still exports normal JS code at the end. It has not yet been adopted by the majority of developers though, so we decided to use

common JS for ixy. Interestingly, ESLint and TypeScript work in very similar ways [37], and using our configuration, we should not run into any issues.

2.3 Low Level Operations

To anyone who uses JS in their daily life, low level might seem the most unfitting description for the language. Because it is mostly used in web applications, there is little to no need for low level operations. However, there are some options JS provides us with. These options mainly consist of the `ArrayBuffer` construct [3], which is essentially just a buffer in memory. JS then provides us with some methods and classes to work on this buffer, such as `DataView` [7] and `TypedArrays` [35]. These classes are similar in the way that they wrap around an `ArrayBuffer` and provide functions to manipulate and read from it.

```
1  const buffer = new ArrayBuffer(8); // 8 byte big buffer
2  const dataView = new DataView(buffer);
3  const typedArray = new Uint8Array(buffer);
4
5  typedArray[2] = 200;
6  console.log(dataView.getUint8(2)); // 200
7  dataView.setUint16(4, 256, true); // use little endian set to true
8  // typedArray is limited to 8-bit operations
9  console.log(typedArray[4]); // 0
10 console.log(typedArray[5]); // 1
```

Listing 2.4: Example of low level JS operations

The main difference, however, is that `DataViews` give us ambiguous access to single read and write operations of bit lengths from 8 to 32, and sometimes even 64 bit. We will get into these edge cases later in Chapter 5. `TypedArrays` on the other hand give us access to an array-like object which can be used just like a normal array in JS and has far better performance than `DataViews`, but it relies on defined bit sizes. In Chapter 5, we will show that the use of both was necessary to develop this driver.

Chapter 3

Node.js as a runtime

3.1 Introduction to Node

The official website does a good job of explaining Node.js (Node) in one single sentence:

> As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications. [25]

Our goal for this thesis is to develop a scalable solution which is - in possibly the most literal way - a so-called network application.

In general, JS is run directly in the browser. Considering the whole JS market, there are no real alternatives or competitors for Node when it comes to running it directly on a machine.

Node uses Chrome V8 (V8) which was originally developed for the Google Chrome browser. The goal of V8 is to make JS run with very high performance by optimizing multiple parts of the runtime.

It compiles JS directly into native machine code before executing it, instead of more traditional techniques such as interpreting bytecode or compiling the whole program to machine code and executing it from a filesystem. The compiled code is additionally optimized (and re-optimized) dynamically at runtime, based on heuristics of the code's execution profile. The optimization techniques used include inlining, elision of expensive runtime properties, and inline caching. The garbage collector is a generational incremental collector. [6]

In Chapter 5, we will discuss in detail how V8 influenced the performance of this driver.

3.2 The Node Eventloop

According to Node.js Foundation's own explanation of Node [25], it is similar in design to, and influenced by, systems like Ruby's Event Machine or Python's Twisted. But it takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems, there is always a blocking call to start the event loop. Typically, behavior is defined by callbacks at the beginning of a script and a blocking call that starts a server at the end. In Node there is no such start-the-event-loop call, it simply enters the event loop after executing the input script. Node exits the event loop when there are no more callbacks to perform. This behavior is like browser JS - the event loop is hidden from the user.

If the reader would like to know more about the event loop and how it works, this 20 minute video by the JSConf [31] is a good starting point.

There was not enough time for us to go further into how we might use asynchronous functions, e.g. for handling asynchronous tasks of the NIC such as sending packets, so this specific part of node is not used to its fullest potential in the current implementation.

3.3 Optimization in Node

Let's take a quick and very rough look at the general optimization JS goes through when using V8 and Node.

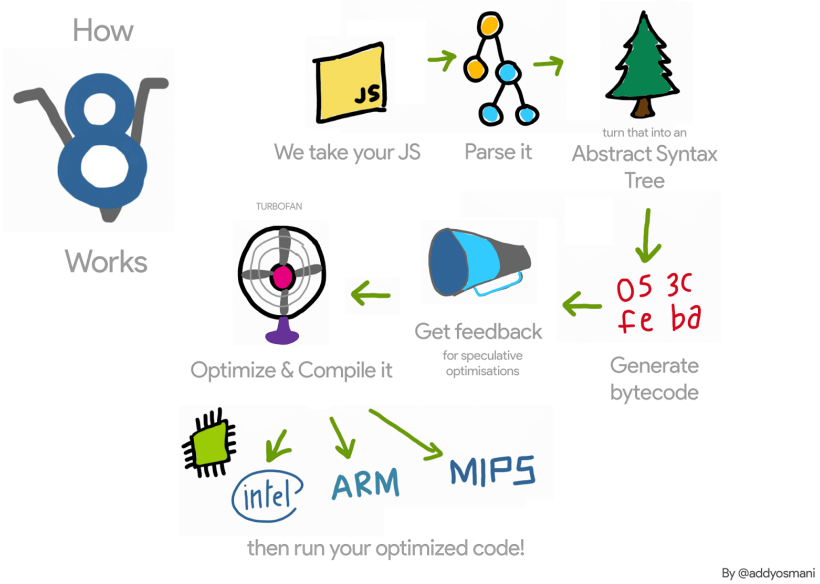


Figure 3.1: Illustration of the V8 optimization process [17]

Put simply, this means that the better V8 can speculate the use of our code, the better it will perform. In the article that included Figure 3.1, the main topic is how V8 optimized lazy unlinking of deoptimized functions. It states some impressive performance upgrades in comparison to older versions, but what would be even more performant than lazy unlinking is if functions never became deoptimized to begin with. So, as long as they remain used and V8 is aware of that, the performance will be at its best. For that reason, using classes to include definitions of frequently used functions is highly performant.

3.4 Low Level Operations

As already mentioned in Chapter 2, low-level JS is very limited but does exist. Node provides some very relevant additions to this. First of all, since JS conforms to the ECMAScript (ES) specification, there are some things it does not yet fully support. Most relevant to us is the type `BigInt`, which represents 64-bit numbers. As of now, `BigInt` is in stage 3 of the ES specification. Once it makes it to stage 4 of the draft, which is the final specification [34], `BigInt` will become the second built-in numeric type in JS. [4] [5] But because it has not been, and currently is not part of ES and therefore common JS, only a few engines support it. Thankfully, it is already included in V8 and therefore in Node.

Because `numbers` in JS are always 64-bit floating point values [9], they have limitations as to what their maximum integer value can be. Since we need 64-bit integers to represent physical addresses, having access to the type `BigInt` is a huge help for the development of this driver. This also means that the aforementioned `DataViews` and `TypedArrays` also support 64-bit representation when used within Node.

Another feature Node provides us with is the possibility to compile C++ and C code directly into Node as an addon and call exported functions from within our JS code using the standard import syntax. This is possible because V8 is written in C++ and the Node team built support for custom addons. [24]

3.5 Using C in JS via N-API

The framework Node provides us with is called Node-API (N-API). It is independent from the underlying JS runtime (for example V8) and is maintained as part of Node itself. This Application Programming Interface (API) is Application Binary Interface stable across versions of Node. [24] It is intended to insulate addons from changes in the underlying JS engine and allow modules compiled for one major version to run on later major versions of Node without recompilation.

Using N-API, we can write C code to access low-level functionalities JS does not offer and include them and the values they return directly in our JS code. Since JS is scarcely used for low-level code though, there are very few examples and resources besides the official documentation. Most of the online resources use the C++ framework Node offers, which works differently than the C part. However, since the original implementation of ixy is written in C, we wanted to keep our code close to it where possible making it easier to compare both and keeping the whole ixy project more consistent. Understanding how to use the N-API and what its API offers took longer than expected, impacting the amount of C code we use. Given more time, we could have probably moved more from C directly into JS. Listing 3.1 shows an example of how N-API is used by one of the functions we implemented using C.

```
napi_value dataviewToPhys(napi_env env, napi_callback_info info)
{
  void *virt;
  napi_status stat;
  size_t sizeOfArray;
  size_t argc = 1;
  size_t byteOffset;
  napi_value argv[1];
  napi_value arrayBuffer;
  stat = napi_get_cb_info(env, info, &argc, argv, NULL, NULL);
  if (stat != napi_ok)
  {
    napi_throw_error(env, NULL, "Failed to parse arguments");
  }
  stat = napi_get_dataview_info(env,
                                argv[0],
                                &sizeOfArray,
                                &virt,
                                &arrayBuffer,
                                &byteOffset);
  if (stat != napi_ok)
  {
    napi_throw_error(env, NULL, "Failed to get virtual Memory from
        Dataview.");
  }
  uintptr_t physPointer = virt_to_phys(virt);
  napi_value ret;
  stat = napi_create_bigint_uint64(env, physPointer, &ret);
  if (stat != napi_ok)
  {
    napi_throw_error(env, NULL, "Failed to write PhysAddr into bigint.");
  }
  return ret;
}
```

Listing 3.1: Example of a function we wrote using N-API

Chapter 4

Implementation

Unless stated otherwise, any code references in the following sections refer to the version 1.0 release of the master branch of our implementation. The logic of the NIC is explained in detail in the Intel datasheet revision 3.3 (March 2016) of the 82599ES datasheet [21].

4.1 Using the C implementation as a guideline

In general, the structure of the Ixy JavaScript implementation (ixy.js) is kept very close to that of the C implementation. The first intention was to ensure an easy start by simply copying the logic of the C code into JS. In the end, we kept most of the similar structure because this makes it easier to read multiple implementations of ixy whilst seeing a rather similar distribution of functionalities. It also holds the possibility to add other NICs and driver specifications without needing to add much besides the core definitions because the logic calls specific implementations that are defined at the lowest level of the driver, which is the definition of the hardware and register manipulation. The only files that are specific to our choice of NIC are `ixgbeDevice.js`, `init.js`, and `queue.js`. Some definition parts in `IXYclass.js`, such as flags to set are also specific, but upon adding new NICs, one can very easily abstract them and call specific definitions depending on the NIC in use. We could have also included the `init` and `queue` functions directly in the `ixgbeDevice` class, but that would be unreadable and too far from the structure C uses.

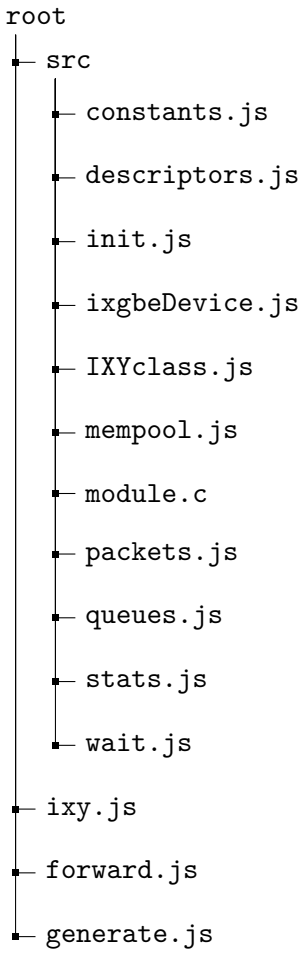


Figure 4.1: Folder structure of our repository

4.2 Driver Design

The goal of ixy is simplicity and speed and since we are working with a very high level language, the simplicity plays a far bigger role in our implementation. We divided the code into multiple sections, which have their own logic and purpose, and only connect them with each other where needed. These different sections are their own files within the `/src` folder as can be seen in Figure 4.1.This makes it easy to read and understand certain parts of the driver, as well as highlighting the connection between them. With a very simple and intuitive API, all users have to do is to look at the functionalities they need, find them in `/src`, import and use them. We showcase this in `ixy.js` with two example programs, `forward.js`, which forwards packets between two pci addresses, and `generate.js`, which generates packets from a single pci address.

Unlike in some other languages, packages and libraries of third parties are well supported and generally should be used where applicable in JS projects. Since this project is very specific and focuses heavily on low level, there are no libraries to help us, though. But if, for example, Node had not supported `BigInt` yet, we could have used a collection of libraries to simulate the functionality for us. There are libraries that could handle commands which exist in other languages, e.g. `mmap` from C, but these libraries are essentially also written in C or C++. Taking a leaner approach and using our own C code whilst not implementing anything we do not need seemed like the best and most readable approach. This way, we do not have a single black box in our code, with every functionality and action being able to be traced back to a line of code we wrote.

The C implementation also adds all the register offset definitions in a file, which we partly ported to JavaScript Object Notation (JSON). We use less than 10% of them, so there was no need to copy all definitions, and as JS uses Just in time compilation (JIT), these definitions cannot be compiled into our code statically, which could impact performance with a large number of options to search through. Our definitions file is about 300 lines long, compared to C one's comprising about 4000 lines. Since we base all the definitions on the datasheet [21], it is very easy to add new definitions if needed to add more functionalities to the driver.

4.3 Security

The paper on the Rust implementation of ixy [11] condenses the risk and a possible solution very nicely:

It is important to note that our implementation requires root privileges to access the Peripheral Component Interconnect Express (PCIe) device. However, this is still better than a custom-built kernel module which by default has root privileges and has to be written in C, an unsafe programming language. In theory, the driver could drop its privileges after initializing the device. Unfortunately, this is not enough since the device is still under full control of the driver and can write to any memory regions through Direct Memory Access (DMA).

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

These additions have been included by the C [14] and Rust [10] implementation already, and the expected benefits have been confirmed.

4.4 Initialization

To prepare the NIC and instantiate a device that is able to send and receive packets, a single call to `IxgbeDevice.init` is needed. It requires the PCIe address of the NIC, the number of receive queues, and the number of transmit queues as parameters.

This first instantiates an `ixgbeDevice` that calls the function `getIXYAddr` during its construction, which is implemented with C and returns a memory buffer for us to use in JS. This buffer is mapped to the provided PCIe address and after creating a 32-bit `TypedArray` wrapped around the buffer we can manipulate the registers the NIC provides us with directly from within JS. It continues by calling our `init.js` exported function, which handles the complete setup of the NIC. First, it disables interrupts, then resets the NIC and afterwards disables interrupts again, because resetting the NIC reactivated that setting. We then make sure the NIC has successfully activated DMA and initialize the link.

As displayed in the following listings, the JS version is a lot easier to read than the one in C. This is also much safer, as we have no way of writing outside of bounds by accident because JS already handles potential errors inside the native class and function prototypes.

```
1 setReg(reg, val) {
2     this.mem32[reg / 4] = val;
3   }
```

Listing 4.1: Function to set registers in JS

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

Listing 4.2: Function to set registers in C

Another difference worth mentioning is that while we are using JS, we have the freedom to also write object-oriented code, which means we can attach those functions directly to the class of the driver and have no need for unnecessarily handing over parameters through functions.

All of this is done by writing to and reading from the `TypedArray` we created earlier. The behavior and registers as well as values we need to do this with are all defined inside the official datasheet [21]. Where relevant, we refer to certain sections of it in comments from within the code.

4.5 DMA allocation

After resetting the statistic counters, we need to initialize the Receiving (RX) and Transmitting (TX) queues to prepare them for sending and receiving packets. For queue initialization, and later for mempool creation, we need to allocate DMA memory. This is because we need memory that is shared independently between our driver, which is executed by the Central Processing Unit (CPU) and the NIC. This means that the NIC will access the memory via physical addresses. The main problem for our implementation is that we cannot actually create or map DMA memory from JS, so we need to do this by using the N-API and C. Within C, there is another problem: we cannot make sure that the Linux kernel does not migrate our mapped memory, which would change the physical address and therefore break the functionality of DMA.

To solve this problem, we allocate huge pages with a size of 2MiB [20]. These pages cannot be migrated by the Linux kernel, which provides us with the safety of non-changing physical addresses. Allocating DMA memory on huge pages is rather simple. The `setup-hugetlbfs.sh` script in our repository creates a `hugetlbfs` mount point and writes the required number of huge pages to a `sysfs` file. Memory allocation in C is then achieved by creating a new file in the mounted directory and mapping the file into memory using `mmap`.

There exist packages for JS to use mmap natively [22], but those are essentially just compiled C or C++ code, so using it directly in C and leaving out unnecessary bloat was the leaner approach to implement DMA. Another bonus is that we have no dependencies for this project. We do use a single package to expose our example implementations to the command line, but it is not needed to run the actual driver.

Inside of this DMA memory we save the descriptors for all the packages a queue can potentially have. The driver and NIC can then both read from and write to the descriptors and therefore exchange information about packets. The descriptors also include physical addresses of the memory the packet data is saved in.

4.6 Memory Pools

The packet data therefore also needs to be saved in DMA memory. Since it would be a huge overhead to allocate new memory from our huge pages every time a packet is received or created, we want to have a Memory Pool (mempool) handle already allocated memory efficiently. Mempools provide a simple interface for receiving and freeing physical addresses for packets by keeping track of their own capacity and the indexes of packets that are free and used. Using this structure, every RX queue has its own mempool to handle incoming packet data. TX queues do not need mempools for handling packet data as it suffices to pass packets from existing pools to them.

In C mempools, packet data includes 64 bits of meta data, which we either do not need to save or save within our JS objects. This also gets rid of a few potentially confusing offsets, e.g. for physical addresses we give the NIC.

Since our hugepages are large enough to encompass multiple different DMA memories, we include the mempools as well as the descriptor buffers the queues use.

4.7 Receiving and sending packets

After the queues are ready to be used we need to actually send and receive packets. Receiving packets works by using a queue of descriptors. It is completely implemented by the NIC, which means that we read the NIC's and write our queue positions from and to the NIC's registers. Since all the packet descriptors are instantiated on the initialization and are never removed, we can simply load their respective data from a certain index to our packet class as shown in Listing 4.3 and then work with it.

We work with packets in batches since writing and reading registers is a relatively costly operation and creates unnecessary overhead. As shown in Chapter 6, the size of the batches does influence performance.

```
1  class Packet {
2    constructor(mempool, index, entry_size) {
3      this.mempool = mempool;
4      this.mempool_idx = index; // we save this for when we free a buffer
5      this.size = 0; // default size, because data is empty
6      // different types of memory access for underlying DMA memory
7      this.mem = new DataView(mempool.base_addr, index * entry_size,
           entry_size);
8      this.mem8 = new Uint8Array(mempool.base_addr, index * entry_size,
           entry_size);
9      this.mem32 = new Uint32Array(mempool.base_addr, index * entry_size,
           entry_size / 4);
10   }
11 }
```

Listing 4.3: Example of packet class in JS

With the packet class as shown in Listing 4.3, we can access all the information including a JS mapped memory buffer of the packet data, which we can use to read from and modify it. After doing everything we wanted to with the packets, we tell the NIC at what index we arrived, which frees the packets descriptor and memory to be used again.

Transmitting packets is a bit more difficult than this, as packets are not instantly sent by the NIC once we give those descriptors free for sending. As this is an asynchronous task, the first idea was to use asynchronous functions inside of JS. But because the NIC side of the process is not written in JS, we could not resolve asynchronous promises which one would commonly use in JS. This means that, just like the C implementation, we have two synchronous loops that skip the packets that were not yet sent and afterwards clean up finished ones. The first loop checks for packets that have been sent by the NIC by reading the index it set for its position in the TX queue and then frees them in the mempool. The second loop does the actual sending, which consists of changing the physical addresses saved in descriptors to point to new data and updating the descriptors size attribute accordingly. We then update the mempool and tell the NIC to send those packets by updating the driver index of the TX queue.

4.8 Ixy.js example implementations

To showcase the functionality and later test performance, we implemented two examples. The first is a simple packet generator, which generates packets with a 60-byte payload and uses one single NIC. The second implementation is a forwarder which uses two NICs and forwards packets between them. The forwarder needs external packets to be sent in, so we use moongen [12] for that.

Using the examples is very simple, as we configured a `package.json` for them. The whole driver can be installed via a simple call of `npm run setup`. To call the examples, we provide `node ixy.js generate pciAddr` and `node ixy.js forward pciAddr1 pciAddr2` with PCIe addresses using the format `0000:03:00.0` .

Chapter 5

Performance Optimization

In addition to a codebase, which is simple to use and understand, another goal was to be as performant as possible. This implies testing performance, which can usually be done using multiple tools. However, for JS there is not that much of a choice.

5.1 Setup

After a bit of research, one can find the library v8-profiler [36], which seemed like the most used and official way to test and record performance of node applications because it has tens of thousands of downloads per week. This number might seem small compared to other widely used JS libraries such as Express.js, which has about nine million weekly downloads [16], but it is the largest we could find for performance testing. After trying to install it, we realized that neither the sources for the installs are available, nor has the package been maintained within the last two years. For this reason, we looked at alternatives and eventually found the flame graph [18] generator tool 0x [1].

Using 0x, we can keep track of the state of the stack so we can see exactly which functions take up most of our execution time. In the description, it states that we can use the linux tool `perf` to add CPU tracing as well, but from Node version 10 and upwards this breaks the export of flame graphs. Because we need to be able to use `BigInts`, we cannot use Node versions prior to v10.4. This limits the information we can get to analyze performance. Despite all this, there is still a lot we can and did learn from looking at the results 0x provided.

All test are done with Node version 10.16.0 unless specified otherwise.

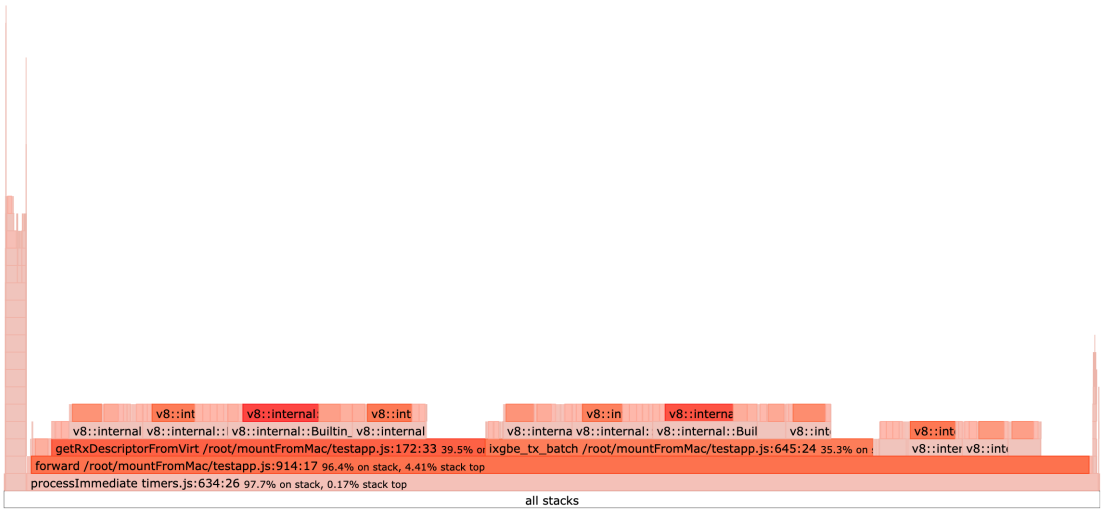


Figure 5.1: Flame graph of first running ixy.js implementation

5.2 Bottlenecks

Analyzing the flame graphs helped us find bottlenecks which were mostly one of the three listed below. After removing these bottlenecks we improved performance from 250 Mbits to more than 1800 Mbits forwarding speed of a single device when running our forward script. That is an improvement of over 600%.

5.2.1 Constructors

As shown in Figure 5.1, a lot of time was spent in V8-internal functions. This is mostly due to constructors, which call the engine directly. We started removing constructors from functions that are called often, e.g. the `rxBatch` function, which is called for receiving packets. These changes improved performance tremendously.

From Figure 5.1 we can see that reading the RX-descriptors from memory via the function `getRxDescriptorFromVirt` took 40% of the execution time. The reason for that is that we constructed `DataViews` inside the function.

To make the driver more performant, we moved the constructors outside into one monolithic constructor and saved offsets inside the descriptor objects. Another change we made was not reading all values on construction of the descriptor but supplying them via functions of the class.

```
class RxDescriptor {
  constructor(virtMem, index = 0) {
    this.memView = new DataView(virtMem, index * 16, 16);
    this.pkt_addr = this.memView.getBigUint64(0, littleEndian);
    // ... and other attributes that work the same way
  }
}
```

Listing 5.1: RxDescriptor before optimization

```
class RxDescriptor {
  constructor(virtMem, index = 0) {
    this.memView = virtMem;
    this.offset = index * 16;
  }
  pkt_addr() { return this.memView.getBigUint64(0 + this.offset,
      littleEndian); }
  // ... and other functions that work the same way
}
```

Listing 5.2: RxDescriptor after moving constructors and read operations

These changes got rid of the whole functions presence in the flame graph and added virtually no overhead, as seen in Figure 5.2. Before the changes a `DataView` was constructed once per packet descriptor that is received during runtime, now it is only called a single time per queue on the initial setup.

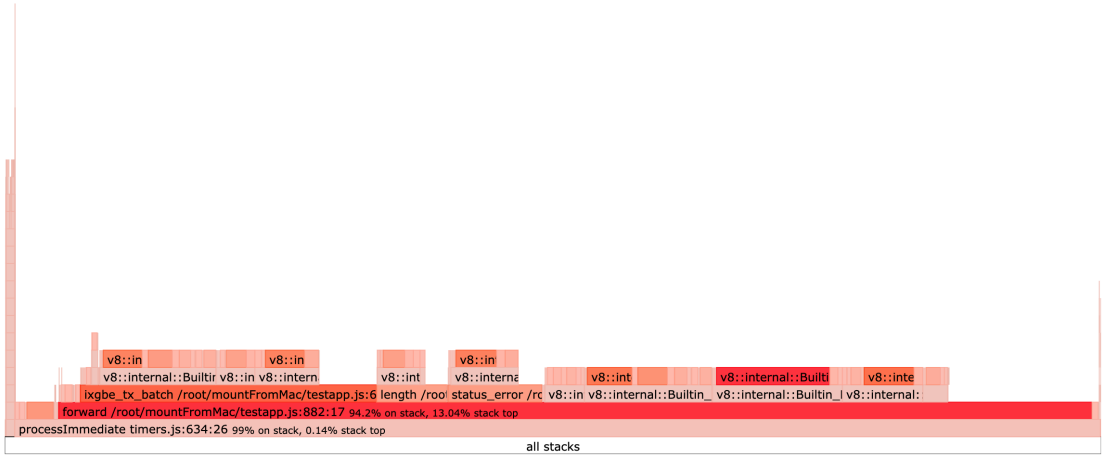


Figure 5.2: Flame graph of ixy.js after first RxDescriptor changes

We also adjusted the TX-descriptor class and optimized its read and write operations before generating the second flame graph, which is why TX has not become the biggest part of the execution time.

The only functions we can see on the stack that are left of `getRxDescriptorFromVirt` are the calls `length` and `status_error`. With most of the other V8-internal functions that are still being called, they represent the second big bottleneck.

5.2.2 DataViews

Most of the V8-internal calls that we can see in the flame graph are the use of `DataView` functions. In Listing 5.1 and Listing 5.2, we can see the use of them. The main syntax used is `dataview.getBitSize(bit offset)` and `dataview.setBitSize(value, bit offset)`. We can also define what endianness [15] to use for sizes bigger than 8 bit. Having a function for 8 bit, 16 bit, 32 bit and even 64 bit when used with Node version 10.4 and later, this is a very convenient and readable approach to handle the low level buffers we provided via DMA.

Because of the many possibilities, it is highly inefficient compared to its counterpart the `TypedArray`, though. According to a V8 blog article on optimizing `DataViews` [19], they take about 128% of the time a `TypedArray` would. This speed has only recently been achieved by the V8 performance update that the blog article is about. Before that `DataViews` took more than 14 times as long.

Since `TypedArrays` are still readable and idiomatic JS, we decided to take some overhead into consideration. For every different bit size we currently used `DataViews` we created a `TypedArray` of that type. The result is that e.g. descriptors is now not a single `DataView` but an object with multiple `TypedArrays`. This is shown in Listing 5.3.

```
1  const queue = {
2        num_entries: defines.NUM_RX_QUEUE_ENTRIES,
3        rx_index: 0,
4        virtual_addresses: new Array(defines.NUM_RX_QUEUE_ENTRIES),
5        descriptors: {
6          d16: new Uint16Array(mem.virt),
7          d32: new Uint32Array(mem.virt),
8          d64: new BigUint64Array(mem.virt),
9        },
10     };
```

Listing 5.3: Queue object from init.js

This also means all functions that worked with `DataViews` before needed to be adjusted. In Listing 5.4, we can see how this changed the syntax. We were able to change almost

all `DataView` uses to `TypedArrays`. The exception is the sequence number of packets because it is a `BigInt` that is not guaranteed to have an offset that is a multiple of 8 bytes. Therefore we cannot use a `BigUint64Array` to access it, as the addressable offset is the size of a `BigInt`, which is 8 bytes.

```
1  class RxDescriptor {
2    constructor(virtMem, index = 0) {
3      this.memView = virtMem;
4      this.offset = index * 16;
5    }
6    pkt_addr() { return this.memView.d64[0 + this.offset / 8]; }
7    upper() {
8      return {
9        status_error: () => this.memView.d32[2 + this.offset / 4],
10       length: () => this.memView.d16[6 + this.offset / 2],
11       vlan: () => this.memView.d16[7 + this.offset / 2],
12     };
13   }
14   // ... and other functions that work the same way
15 }
```

Listing 5.4: RxDescriptor after full optimization

The code is still very structured and readable, and apart from some initial constructor and code line overhead there is no negative impact whatsoever. If we look at the newly created flame graph in Figure 5.3, nearly all of the V8-native functions fell away. `TypedArrays` have far less impact on performance and are not visible in the flame graph by themselves.

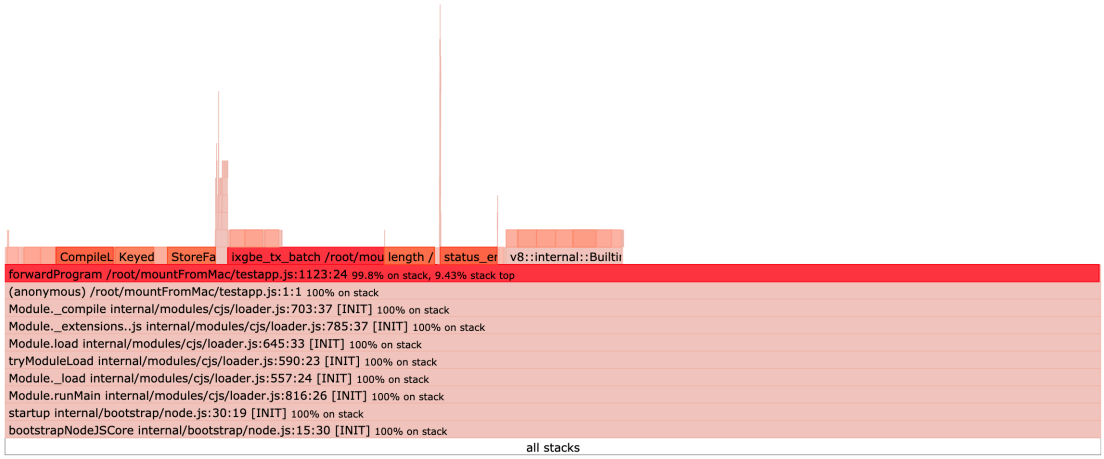


Figure 5.3: Flame graph of ixy.js after DataView to TypedArray changes

But there is still a rather big chunk with over 10% of the execution time left which calls V8-internal code.

5.2.3 BigInt

The aforementioned internal calls are made by the `BigInt` constructor. We scarcely use `BigInts`, but where we do we initially simply called constructors with regular JS `numbers` as input.

In the following code examples, we show one of our uses of `BigInts`. We need to shift a physical address which is a 64-bit sized `BigInt` by 32 bits to get only the second half of it. After that we typecast it into a `number` to be able to use it as a parameter for 32-bit operations. As `BigInts` are not part of the JS type `number`, we cannot bit shift them with `numbers`, but need to use use `BigInts` for the shifting as well [32].

```
1 const PhysEndingSlow = Number(mem.phy >> BigInt(32));
2
3 const PhysEndingFast = Number(mem.phy >> 32n);
```

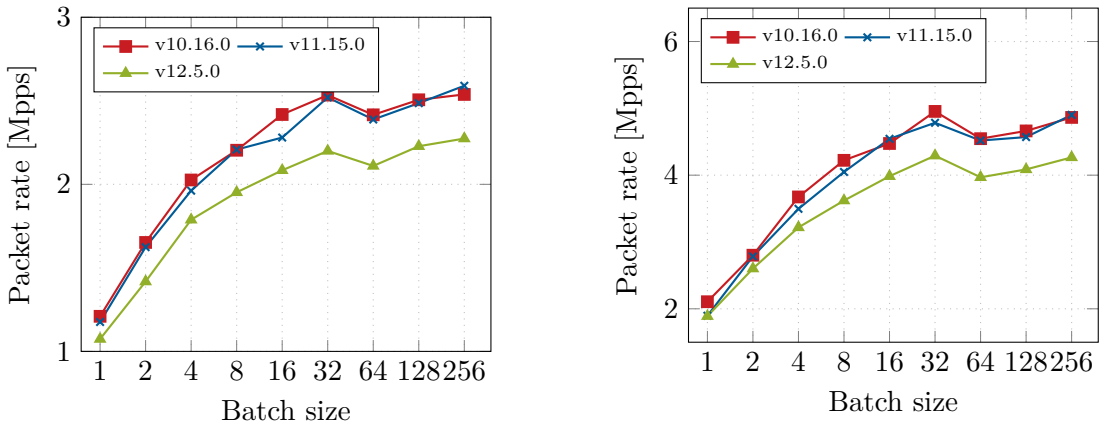
Listing 5.5: Defining BigInts before and after optimization

By changing `BigInts` from calling the constructor to defining them directly in code we completely removed that part from the flame graph and improved performance by over 5%. To put this into perspective, it must be pointed out that we use `BigInts` exactly five times in our entire JS code.

5.3 Difference between Node versions

As Node has a rather interesting release cycle [26], it is useful for us to know the performance different versions have for our code. As of the writing of this thesis version 10 is the active Long-term support (LTS), with version 12 coming up soon to be the next active version. The LTS of version 11 has just been discontinued as it is an odd-numbered release.

As one can take from Figure 5.4, v10 performs best with v11 being very close. V12 seems to perform worse in every possible setting, which might just be because of its status of not being active yet. In conclusion, using v10 because of performance reasons and because of LTS is currently the best decision to make.



(a) Node versions at 1.6 GHz.

(b) Node versions at 3.3 GHz.

Figure 5.4: Performance of different node versions running bidirectional single-core forwarding with varying batch size.

Chapter 6

Evaluation

6.1 Performance

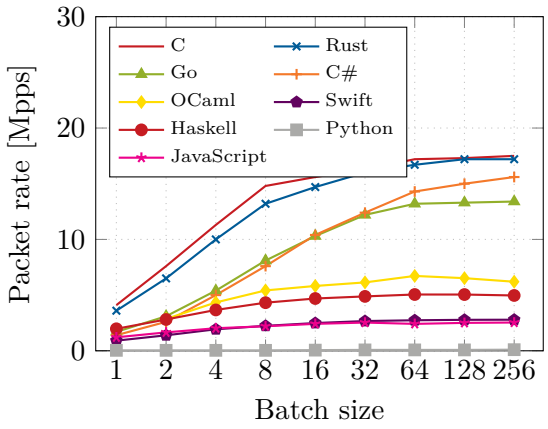


Figure 6.1: Bidirectional single-core forwarding performance with varying batch size at 1.6 GHz

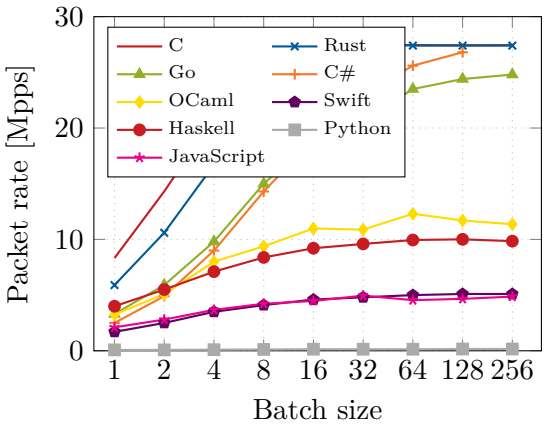


Figure 6.2: Bidirectional single-core forwarding performance with varying batch size at 3.3 GHz

Looking at Figure 6.1 and Figure 6.2, we can see that JS performs rather poorly in comparison with languages such as C and Rust. Our performance is almost identical to the performance of the Swift implementation. The main difference is that JS has its best performance at a batch size of 32 and drops performance with higher batch sizes, which Swift does not. At 3.3 GHz and a batch size of 32 or lower, JS actually performs slightly better than Swift.

In general, larger batches remove overhead and therefore add performance whilst also adding delay and more cache misses. Because of this, most implementations of ixy see 64 as the optimal batch size as the performance tends to get its last big boost from 32 to 64. When deciding on our implementations sweet spot for performance and delay, a batch size of 32 is the obvious choice.

Noteworthy is that every language listed except JS and Python are compiled. That JS is able to have the same performance as a compiled implementation and has far better performance than the only other non-compiled language shows just how well executed Node and our performance optimizations are.

6.2 Development

As mentioned in Chapter 2, it is easy to prototype and develop in JS. Although we do have a rather large part of C code, most of the driver is written in idiomatic JS. The development is convenient because of a large support of addons and plugins for formatting and structure such as ESLint. In addition to that, since JS is the most referenced language on Stack Overflow [29], the community helps fix any problems one might encounter.

In addition to that, we used less code than C. The C implementation has about 1000 lines of code after subtracting the NIC definitions and example programs. Based on the same conditions, our code has around 900 lines of code with about 650 lines of JS.

On the other hand, since we were getting into low level territory, there were not remotely as many users or resources to help us. Developing in JS is definitely more comfortable than in C, but if one has a problem virtually nobody has ever encountered because nobody else uses the language for that kind of applications, it can become difficult.

6.3 JavaScript's suitability for low level development

After understanding N-API well enough, Node suprisingly opened up a whole new world of low level JS to work with. JS is not designed to do low level development, but it already has types and functions to handle buffers and all types of bit operations, and produces very clean and readable code. Node on the other hand is designed with low-level operations in mind, opening the runtime to custom addons for any use cases we cannot complete with common JS.

Memory management is one of the biggest problems with low level JS. We needed functions in C to allocate DMA memory and get physical addresses. The four C functions

we added amount to about 250 lines of C, but a large part of that are N-API wrappers to expose them to Node.

Chapter 7

Related Work

As mentioned earlier, JS is rarely used for low-level applications. However, there are some projects that tackle this subject. The projects that are closest to this thesis' topic are NodeOS [27] and OS.js [30].

7.1 NodeOS

NodeOS is a full operating system built on top of the linux kernel. The latest release is from June 2017 [28], but there are still changes being made to the Github repositories.

It runs on Node, and therefore, in user space. Because of that, it also implements user-space drivers, including network-related ones [33]. These are completely written in C++ and then compiled into Node addons. Based on this, we can assume that if there were a network driver implementation, which we could not find, it would also be written in C++ and not idiomatic JS. Therefore, it is not comparable to ixy.js.

7.2 OS.js

OS.js is an open-source JS web-desktop implementation for browsers. It includes a window manager, application APIs, Graphical User Interface toolkits and filesystem abstraction. The implementation is split into client-side code, which runs in the browser, and server-side code, which runs on Node. Since the server side has a complete operating system running below Node, there is no reason to write user-space drivers for features that are already being handled. OS.js does not aim to be a highly efficient user-space-driven operating system, but to be an operating system that is accessible over the web

and as fully functional as possible. Thus, OS.js does not implement user-space network drivers and is not comparable to ixy.js, either.

Based on the findings of our research in the course of this thesis, ixy.js is the first and only implementation of a user-space network driver using idiomatic JS.

Chapter 8

Conclusion

Since network drivers are the type of drivers that heavily rely on performance and JS is not the greatest in that category by any means, it does not qualify as the go-to language to write network drivers.

For writing a driver that needs to be readable, understandable and accessible, to understand how drivers work and for similar educational purposes, it does, however, work very well.

Additionally, as not every driver needs to be highly performant, JS could potentially be used for other user-space drivers that could then be used in production environments.

Future work on the driver could consist of multiple things such as adding virtIO support as well as the usage of an IOMMU, which has both already been done by the C implementation. Another area to work on would be to revise the C we use, because there is still room to optimize the usage of C by moving logic into JS. The last big continuation of this driver implementation could be the typing, which would entail a TypeScript version of ixy.js. This would take care of some of the concerns during development and production use of drivers, such as runtime errors, as well as make the code even more readable.

All things considered, we successfully showed that writing user-space drivers is possible and feasable with JS and therefore virtually any programming language, as JS was most definitely not intended for such endeavors.

Chapter A

List of acronyms

API	Application Programming Interface.
CPU	Central Processing Unit.
DMA	Direct Memory Access.
ES	ECMAScript. A scripting-language specification standardized by Ecma International in ECMA-262 and ISO/IEC 16262. It was created to standardize JavaScript, so as to foster multiple independent implementations.
ixy	ixy-languages project. A collection of simple user-space packet processing frameworks for educational purposes in multiple programming languages. ixy takes exclusive control of a network adapter and implements the whole driver in userspace.
ixy.js	ixy JavaScript implementation. A simple user-space packet processing framework for educational purposes written in JavaScript. Also the product of this thesis.
JIT	Just in time compilation.
JS	JavaScript.
JSON	JavaScript Object Notation. An open-standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value).
LTS	long-term support.

N-API Node-API. N-API (pronounced N as in the letter, followed by API) is an API for building native addons. It is independent from the underlying JavaScript runtime (for example V8) and is maintained as part of Node.js itself. This API is Application Binary Interface (ABI) stable across versions of Node.js. It is intended to insulate addons from changes in the underlying JavaScript engine and allow modules compiled for one major version to run on later major versions of Node.js without recompilation.

NIC Network Interface Card.

Node Node.js. An asynchronous event driven JavaScript runtime, designed to build scalable network applications.

PCIe Peripheral Component Interconnect Express. A high-speed serial computer expansion bus standard, designed to replace the older PCI, PCI-X and AGP bus standards. It is the common motherboard interface for personal computers' graphics cards, hard drives, SSDs, Wi-Fi and Ethernet hardware connections.

RX receiving.

TX transmitting.

V8 Chrome V8. An open-source JavaScript engine developed by The Chromium Project for Google Chrome and Chromium web browsers.

Bibliography

[1] *0x package*. Apr. 2019. URL: https://www.npmjs.com/package/0x.

[2] *Airbnb JavaScript Style Guide*. July 2019. URL: https://github.com/airbnb/javascript.

[3] *ArrayBuffer Documentation*. Mar. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer.

[4] *BigInt Documentation*. July 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt.

[5] *BigInt proposal Github repository*. June 2019. URL: https://github.com/tc39/proposal-bigint.

[6] *Chrome V8*. July 2019. URL: https://en.wikipedia.org/wiki/Chrome_V8.

[7] *DataView Documentation*. Mar. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView.

[8] "Developer Survey Results 2019". In: *Stack Overflow* (2019). URL: https://insights.stackoverflow.com/survey/2019#technology-_-programming-scripting-and-markup-languages.

[9] *ECMAScript® 2020 Language Specification*. July 2019. URL: https://tc39.es/ecma262/#sec-number-constructor-number-value.

[10] Simon Ellmann. *ixy.rs Github repository*. July 2019. URL: https://github.com/ixy-languages/ixy.rs.

[11] Simon Ellmann. "Writing Network Drivers in Rust". Bachelors Thesis. Technical University Munich, Oct. 2018.

[12] Paul Emmerich. *benchmark-scripts Github repository*. Oct. 2018. URL: https://github.com/ixy-languages/benchmark-scripts.

[13] Paul Emmerich. *ixy-languages Github repository*. June 2019. URL: https://github.com/ixy-languages/ixy-languages.

[14] Paul Emmerich. *ixy.c Github repository*. June 2019. URL: https://github.com/emmericp/ixy.

[15] *Endianness*. June 2019. URL: https://en.wikipedia.org/wiki/Endianness.

[16] *express.js package*. June 2019. URL: https://www.npmjs.com/package/express.

[17] Juliana Franco. "An internship on laziness: lazy unlinking of deoptimized functions". In: *V8 dev blog* (Oct. 2017). URL: https://v8.dev/blog/lazy-unlinking.

[18] Brendan Gregg. *CPU Flame Graphs*. Mar. 2017. URL: http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

[19] Théotime Grohens and Benedikt Meurer. "Improving DataView performance in V8". In: *V8 dev blog* (Sept. 2018). URL: https://v8.dev/blog/dataview.

[20] *Hugetlbpage Documentation*. July 2019. URL: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[21] *Intel® 82599 10 GbE Controller Datasheet*. Mar. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[22] *jsmmap package*. Apr. 2019. URL: https://www.npmjs.com/package/mmap-io.

[23] *List of programming languages*. July 2019. URL: https://en.wikipedia.org/wiki/List_of_programming_languages.

[24] *N-API Documentation*. July 2019. URL: https://nodejs.org/api/n-api.html#n_api_n_api.

[25] *Node.js About*. Aug. 2018. URL: https://nodejs.org/en/about/.

[26] *Node.js Releases*. July 2019. URL: https://nodejs.org/en/about/releases/.

[27] *NodeOS*. Sept. 2018. URL: http://node-os.com/.

[28] *NodeOS releases*. June 2017. URL: https://github.com/NodeOS/NodeOS/releases.

[29] Stephen O'Grady. "The RedMonk Programming Language Rankings: January 2019". In: *Redmonk.com* (Mar. 2019). URL: https://redmonk.com/sogrady/2019/03/20/language-rankings-1-19/.

[30] *OS.js*. July 2019. URL: https://www.os-js.org/.

[31] Philip Roberts. *What the heck is the event loop anyway?* Oct. 2014. URL: https://www.youtube.com/watch?v=8aGhZQkoFbQ.

[32] *Should shifts of BigInt by Number be allowed?* May 2017. URL: https://github.com/tc39/proposal-bigint/issues/40.

[33] *Sockios*. Mar. 2016. URL: https://github.com/NodeOS/sockios.

[34] *The TC39 Process*. July 2019. URL: https://tc39.es/process-document/.

[35] *TypedArrays documentation*. Mar. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays.

[36] *v8-profiler*. Mar. 2017. URL: https://www.npmjs.com/package/v8-profiler.

[37] *What are ESLint and TypeScript, and how do they compare?* July 2019. URL: https://github.com/typescript-eslint/typescript-eslint#what-are-eslint-and-typescript-and-how-do-they-compare.