User Space Network Drivers

Paul Emmerich
Technical University of Munich
emmericp@net.in.tum.de

Maximilian Pudelko
Technical University of Munich
Open Networking Foundation
max@opennetworking.org

Simon Bauer
Technical University of Munich
bauersi@net.in.tum.de

Stefan Huber
Technical University of Munich
hubestef@net.in.tum.de

Thomas Zwickl
Technical University of Munich
zwickl@net.in.tum.de

Georg Carle
Technical University of Munich
carle@net.in.tum.de

ABSTRACT

The rise of user space packet processing frameworks like DPDK and netmap makes low-level code more accessible to developers and researchers. Previously, driver code was hidden in the kernel and rarely modified—or even looked at—by developers working at higher layers. These barriers are gone nowadays, yet developers still treat user space drivers as black-boxes magically accelerating applications. We want to change this: every researcher building high-speed network applications should understand the intricacies of the underlying drivers, especially if they impact performance. We present ixy, a user space network driver designed for simplicity and educational purposes to show that fast packet IO is not black magic but careful engineering. ixy focuses on the bare essentials of user space packet processing: a packet forwarder including the whole NIC driver uses less than 1,000 lines of C code.

This paper is partially written in tutorial style on the case study of our implementations of drivers for both the Intel 82599 family and for virtual VirtIO NICs. The former allows us to reason about driver and framework performance on a stripped-down implementation to assess individual optimizations in isolation. VirtIO support ensures that everyone can run it in a virtual machine.

Our code is available as free and open source under the BSD license at https://github.com/emmericp/ixy.

KEYWORDS

Tutorial, performance evaluation, DPDK, netmap

ANCS'19, 2019

1 INTRODUCTION

Low-level packet processing on top of traditional socket APIs is too slow for modern requirements and was therefore often done in the kernel in the past. Two examples for packet forwarders utilizing kernel components are Open vSwitch [40] and the Click modular router [32]. Writing kernel code is not only a relatively cumbersome process with slow turnaround times, it also proved to be too slow for specialized applications. Open vSwitch was since extended to include DPDK [7] as an optional alternative backend to improve performance [36]. Click was ported to both netmap [41] and DPDK for the same reasons [2]. Other projects also moved kernel-based code to specialized user space code [24, 43].

Developers and researchers often treat DPDK as a black-box that magically increases speed. One reason for this is that DPDK, unlike netmap and others, does not come from an academic background. It was first developed by Intel and then moved to the Linux Foundation in 2017 [27]. This means that there is no academic paper describing its architecture or implementation. The netmap paper [41] is often used as surrogate to explain how user space packet IO frameworks work in general. However, DPDK is based on a completely different architecture than seemingly similar frameworks.

Abstractions hiding driver details from developers are an advantage: they remove a burden from the developer. However, all abstractions are leaky, especially when performance-critical code such as high-speed networking applications are involved. We therefore believe it is crucial to have at least some insights into the inner workings of drivers when developing high-speed networking applications.

We present ixy, a user space packet framework that is architecturally similar to DPDK [7] and Snabb [15]. Both use full user space drivers, unlike netmap [41], PF_RING [34], PFQ [4], or similar frameworks that rely on a kernel driver. ixy is designed for educational use only, i.e., you are meant to use it to understand how user space packet frameworks and drivers work, not to use it in a production environment. Our whole architecture, described in Section 3, aims at simplicity

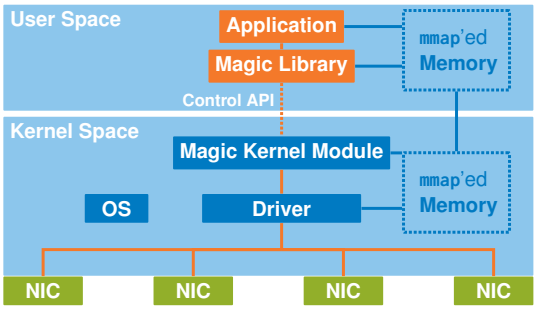

Figure 1: Architecture of user space packet processing frameworks using an in-kernel driver, e.g., netmap, PF_RING ZC, or PFQ.

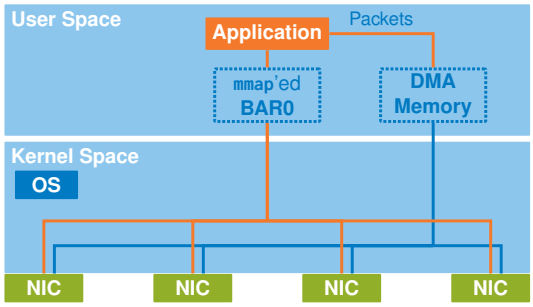

Figure 2: Architecture of full user space network drivers, e.g., DPDK, Snabb, or ixy.

and is trimmed down to the bare minimum. We currently support the Intel ixgbe family of NICs (cf. Section 4) and virtual VirtIO NICs (cf. Section 6). A packet forwarding application is less than 1,000 lines of C code including the whole poll-mode driver, the implementation is discussed in Section 4. It is possible to read and understand drivers found in other frameworks, but ixy's driver is at least an order of magnitude simpler than other implementations. For example, DPDK's implementation of the ixgbe driver needs 5,400 lines of code just to handle receiving and sending packets in a highly optimized way, offering multiple paths using different vector SIMD instruction sets. ixy's receive and transmit path for the same driver is only 127 lines of code.

It is not our goal to support every conceivable scenario, hardware feature, or optimization. We aim to provide an educational platform for experimentation with driver-level features or optimizations. ixy is available under the BSD license for this purpose [38].

2 BACKGROUND AND RELATED WORK

A multitude of packet IO frameworks have been built over the past years, each focusing on different aspects. They can be broadly categorized into two categories: those relying on a driver running in the kernel (Figure 1) and those that re-implement the whole driver in user space (Figure 2).

Examples for the former category are netmap [41], PF_RING ZC [34], PFQ [4], and OpenOnload [44]. They all use the default driver (sometimes with small custom patches) and an additional kernel component that provides a fast interface based on memory mapping for the user space application. Packet IO is still handled by the kernel driver here, but the driver is attached to the application directly instead of the kernel datapath, see Figure 1. This has the advantage that integrating existing kernel components or forwarding packets to the default network stack is feasible with these frameworks. By default, these applications still provide an application with exclusive access to the NIC. Parts of the NIC can often still be controlled with standard tools like `ethtool` to configure packet filtering or queue sizes. However, hardware features are often poorly supported, e.g., netmap lacks support for most offloading features [11].

Note that none of these two advantages is superior to the other, they are simply different approaches for a similar problem. Each solution comes with unique advantages and disadvantages depending on the exact use case.

netmap [41] and XDP [23] are good examples of integrating kernel components with specialized applications. netmap (a standard component in FreeBSD and also available on Linux) offers interfaces to pass packets between the kernel network stack and a user space app, it can even make use of the kernel's TCP/IP stack with StackMap [46]. Further, netmap supports using a NIC with both netmap and the kernel simultaneously by using hardware filters to steer packets to receive queues either managed by netmap or the kernel [3]. XDP is technically not a user space framework: the code is compiled to eBPF which is run by a JIT in the kernel, this restricts the choice of programming language to those that can target eBPF bytecode (typically a restricted subset of C is used). It is a default part of the Linux kernel nowadays and hence very well integrated. It is well-suited to implement firewalls that need to pass on traffic to the network stack [14]. More complex applications can be built on top of it with AF_XDP sockets, resulting in an architecture similar to netmap applications. Despite being part of the kernel, XDP does not yet work with all drivers as it requires a new memory model for all supported drivers. At the time of writing, XDP in kernel 4.19 (current LTS) supports fewer drivers than DPDK [5, 22] and does not support forwarding to different NICs.

DPDK [7], Snabb [15], and ixy implement the driver completely in user space. DPDK still uses a small kernel module with some drivers, but it does not contain driver logic and is only used during initialization. Snabb and ixy require no kernel code at all, see Figure 2. A main advantage of the full user space approach is that the application has full control

over the driver leading to a far better integration of the application with the driver and the hardware. DPDK features the largest selection of offloading and filtering features of all investigated frameworks [6]. The downside is the poor integration with the kernel, DPDK's KNI (kernel network interface) needs to copy packets to pass them to the kernel unlike XDP or netmap which can just pass a pointer. Other advantages of DPDK are its support in the industry, mature code base, and large community. DPDK supports virtually all NICs commonly found in servers [5], far more than any other framework we investigated here.

ixy is a full user space driver as we want to explore writing drivers and not interfacing with existing drivers. Our architecture is based on ideas from both DPDK and Snabb. The initialization and operation without loading a driver is inspired by Snabb, the API based on explicit memory management, batching, and driver abstraction is similar to DPDK.

3 DESIGN

Function names and line numbers referring to our implementation are hyperlinked to the source code on GitHub.

The language of choice for the explanation here and initial implementation is C as the lowest common denominator of systems programming languages. Implementations in other languages are also available [8]. Our design goals are:

- Simplicity. A forwarding application including a driver should be less than 1,000 lines of C code.
- No dependencies. One self-contained project including the application and driver.
- Usability. Provide a simple-to-use interface for applications built on it.
- Speed. It should be reasonable fast without compromising simplicity, find the right trade-off.

It should be noted that the Snabb project [15] has similar design goals; ixy tries to be one order of magnitude simpler. For example, Snabb targets 10,000 lines of code [25], we target 1,000 lines of code and Snabb builds on Lua with LuaJIT instead of C limiting accessibility.

3.1 Architecture

ixy only features one abstraction level: it decouples the used driver from the user's application. Applications call into ixy to initialize a network device by its PCI address, ixy choses the appropriate driver automatically and returns a struct containing function pointers for driver-specific implementations. We currently expose packet reception, transmission, and device statistics to the application. Packet APIs are based on explicit allocation of buffers from specialized *memory pool* data structures.

Applications include the driver directly, ensuring a quick turn-around time when modifying the driver. This means that the driver logic is only a single function call away from the application logic, allowing the user to read the code from a top-down level without jumping between complex abstraction interfaces or even system calls.

3.2 NIC Selection

ixy is based on custom user space re-implementation of the Intel ixgbe driver and the VirtIO virtio-net driver cut down to their bare essentials. We tested our ixgbe driver on Intel X550, X540, and 82599ES NICs, virtio-net on qemu with and without vhost, and on VirtualBox. All other frameworks except DPDK are also restricted to very few NIC models (typically 3 or fewer families) and ixgbe is (except for OpenOnload only supporting their own NICs) always supported.

We chose ixgbe for ixy because Intel releases extensive datasheets and the ixgbe NICs are commonly found in commodity servers. These NICs are also interesting because they expose a relatively low-level interface to the drivers. Other NICs like the newer Intel XL710 series or Mellanox ConnectX-4/5 follow a more firmware-driven design: a lot of functionality is hidden behind a black-box firmware running on the NIC and the driver merely communicates via a message interface with the firmware which does the hard work. This approach has obvious advantages such as abstracting hardware details of different NICs allowing for a simpler more generic driver. However, our goal with ixy is understanding the full stack—a black-box firmware is counterproductive here and we have no plans to add support for such NICs.

VirtIO was selected as second driver to ensure that everyone can run the code without hardware dependencies. A second interesting characteristic of VirtIO is that it is based on PCI instead of PCIe, requiring a different approach to implement the driver in user space.

3.3 User Space Drivers in Linux

There are two subsystems in Linux that enable user space drivers: `uio` and `vfio`, we support both.

`uio` exposes all necessary interfaces to write full user space drivers via memory mapping files in the `sysfs` pseudo filesystem. These file-based APIs give us full access to the device without needing to write any kernel code. ixy unloads any kernel driver for the given PCI device to prevent conflicts, i.e., there is no driver loaded for the NIC while ixy is running.

`vfio` offers more features: IOMMU and interrupts are only supported with `vfio`. However, these features come at the cost of additional complexity: It requires binding the PCIe device to the generic `vfio-pci` driver and it then exposes an API via `ioctl` syscalls on special files.

One needs to understand how a driver communicates with a device to understand how a driver can be written in user space. A driver can communicate via two ways with a PCIe

device: The driver can initiate an access to the device's Base Address Registers (BARs) or the device can initiate a direct memory access (DMA) to access arbitrary main memory locations. BARs are used by the device to expose configuration and control registers to the drivers. These registers are available either via memory mapped IO (MMIO) or via x86 IO ports depending on the device, the latter way of exposing them is deprecated in PCIe.

3.3.1 Accessing Device Registers. MMIO maps a memory area to device IO, i.e., reading from or writing to this memory area receives/sends data from/to the device. `uio` exposes all BARs in the `sysfs` pseudo filesystem, a privileged process can simply `mmap` them into its address space. `vfio` provides an `ioctl` that returns memory mapped to this area. Devices expose their configuration registers via this interface where normal reads and writes can be used to access registers. For example, ixgbe NICs expose all configuration, statistics, and debugging registers via the BAR0 address space. Our implementations of these mappings are in `pci_map_resource()` in `pci.c` and in `vfio_map_region()` in `libixy-vfio.c`.

VirtIO (in the version we are implementing for compatibility with VirtualBox) is unfortunately based on PCI and not on PCIe and its BAR is an IO port resource that must be accessed with the archaic `IN` and `OUT` x86 instructions requiring IO privileges. Linux can grant processes the necessary privileges via `ioperm(2)` [16], DPDK uses this approach for their VirtIO driver. We found it too cumbersome to initialize and use as it requires either parsing the PCIe configuration space or text files in `procfs` and `sysfs`. Linux `uio` also exposes IO port BARs via `sysfs` as files that, unlike their MMIO counterparts, cannot be `mmap`ed. These files can be opened and accessed via normal read and write calls that are then translated to the appropriate IO port commands by the kernel. We found this easier to use and understand but slower due to the required syscall. See `pci_open_resource()` in `pci.c` and `read/write_ioX()` in `device.h` for the implementation.

A potential pitfall is that the exact size of the read and writes are important, e.g., accessing a single 32 bit register with 2 16 bit reads will typically fail and trying to read multiple small registers with one read might not be supported. The exact semantics are up to the device, Intel's ixgbe NICs only expose 32 bit registers that support partial reads (except clear-on-read registers) but not partial writes. VirtIO uses different register sizes and specifies that any access width should work in the mode we are using [35], in practice only aligned and correctly sized accesses work reliably.

3.3.2 DMA in User Space. DMA is initiated by the PCIe device and allows it to read/write arbitrary physical addresses. This is used to access packet data and to transfer the DMA descriptors (pointers to packet data) between driver and NIC. DMA needs to be explicitly enabled for a device via the PCI configuration space, our implementation is in `enable_dma()` in `pci.c` for `uio` and in `vfio_enable_dma()` in `libixy-vfio.c` for `vfio`. DMA memory allocation differs significantly between `uio` and `vfio`.

uio DMA memory allocation. Memory used for DMA transfer must stay resident in physical memory. `mlock(2)` [26] can be used to disable swapping. However, this only guarantees that the page stays backed by memory, it does not guarantee that the physical address of the allocated memory stays the same. The linux page migration mechanism can change the physical address of any page allocated by the user space at any time, e.g., to implement transparent huge pages and NUMA optimizations [28]. Linux does not implement page migration of explicitly allocated huge pages (2 MiB or 1 GiB pages on x86). ixy therefore uses huge pages which also simplify allocating physically contiguous chunks of memory. Huge pages allocated in user space are used by all investigated full user space drivers, but they are often passed off as a mere performance improvement [21, 42] despite being crucial for reliable allocation of DMA memory.

The user space driver hence also needs to be able to translate its virtual addresses to physical addresses, this is possible via the `procfs` file `/proc/self/pagemap`, the translation logic is implemented in `virt_to_phys()` in `memory.c`.

vfio DMA memory allocation. The previous DMA memory allocation scheme is specific to a quirk in Linux on x86 and not portable. `vfio` features a portable way to allocate memory that internally calls `dma_alloc_coherent()` in the kernel like an in-kernel driver would. This syscall abstracts all the messy details and is implemented in our driver in `vfio_map_dma()` in `libixy-vfio.c`. It requires an IOMMU and configures the necessary mapping to use virtual addresses for the device.

DMA and cache coherency. Both of our implementations require a CPU architecture with cache-coherent DMA access. Older CPUs might not support this and require explicit cache flushes to memory before DMA data can be read by the device. Modern CPUs do not have that problem. In fact, one of the main enabling technologies for high speed packet IO is that DMA accesses do not actually go to memory but to the CPU's cache on any recent CPU architecture.

3.3.3 Interrupts in User Space. `vfio` features full support for interrupts, `vfio_setup_interrupt()` in `libixy-vfio.c` enables a specific interrupt for vfio and associates it with an eventfd file descriptor. `enable_interrupt()` in `ixgbe.c` configures the interrupts for a queue on the device.

Interrupts are mapped to a file descriptor on which the usual syscalls like `epoll` are available to sleep until an interrupt occurs, see `vfio_epoll_wait()` in `libixy-vfio.c`.

3.4 Memory Management

ixy builds on an API with explicit memory allocation similar to DPDK which is a very different approach from netmap [41] that exposes a replica[1] of the NIC's ring buffer to the application. Memory allocation for packets was cited as one of the main reasons why netmap is faster than traditional in-kernel processing [41]. Hence, netmap lets the application handle memory allocation details. Many forwarding cases can then be implemented by simply swapping pointers in the rings. However, more complex scenarios where packets are not forwarded immediately to a NIC (e.g., because they are passed to a different core in a pipeline setting) do not map well to this API and require adding manual buffer management on top of this API. Further, a ring-based API is very cumbersome to use compared to one with memory allocation.

It is true that memory allocation for packets is a significant overhead in the Linux kernel, we have measured a per-packet overhead of 100 cycles[2] when forwarding packets with Open vSwitch on Linux for allocating and freeing packet memory (measured with `perf`). This overhead is almost completely due to (re-)initialization of the kernel `sk_buff` struct: a large data structure with a lot of metadata fields targeted at a general-purpose network stack. Memory allocation in ixy (with minimum metadata required) only adds an overhead of 30 cycles/packet, a price that we are willing to pay for the gained simplicity in the user-facing API.

ixy's API is the same as DPDK's API when it comes to sending and receiving packets and managing memory. It can best be explained by reading the example applications `ixy-fwd.c` and `ixy-pktgen.c`. The transmit-only example `ixy-pktgen.c` creates a *memory pool*, a fixed-size collection of fixed-size packet buffers and pre-fills them with packet data. It then allocates a batch of packets from this pool, adds a sequence number to the packet, and passes them to the transmit function. The transmit function is asynchronous: it enqueues pointers to these packets, the NIC fetches and sends them later. Previously sent packets are freed asynchronously in the transmit function by checking the queue for sent packets and returning them to the pool. This means that a packet buffer cannot be re-used immediately, the `ixy-pktgen` example looks therefore quite different from a packet generator built on a classic socket API.

The forward example `ixy-fwd.c` can avoid explicit handling of memory pools in the application: the driver allocates a memory pool for each receive ring and automatically allocates packets. Allocation is done by the packet reception function, freeing is either handled in the transmit function as before or by dropping the packet explicitly if the output link is full. Exposing the rings directly similar to netmap could significantly speed up this simple example application at the cost of usability.

3.5 Security Considerations

User space drivers effectively run with root privileges even if they drop privileges after initializing devices: they can use the device's DMA capabilities to access arbitrary memory locations, negating some of the security advantages of running in user space. This can be mitigated by using the IO memory management unit (IOMMU) to isolate the address space accessible to a device at the cost of an additional (hardware-accelerated) lookup in a page table for each memory access by the device.

IOMMUs are available on CPUs offering hardware virtualization features as they were designed to pass PCIe devices (or parts of them via SR-IOV) directly into VMs in a secure manner. Linux abstracts different IOMMU implementations via the `vfio` framework which is specifically designed for "safe non-privileged userspace drivers" [29] beside virtual machines. Our `vfio` backend allows running the driver and application as an unprivileged user. Of the investigated other frameworks only netmap supports this. DPDK also offers a `vfio` backend and has historically supported running with unprivileged users, but recent versions no longer support this with most drivers. Snabb's `vfio` backend was removed because of the high maintenance burden and low usage.

4 IXGBE IMPLEMENTATION

All page numbers and section numbers for the Intel datasheet refer to revision 3.3 (March 2016) of the 82599ES datasheet [20]. Function names and line numbers referring to our implementation are hyperlinked to the source code on GitHub.

ixgbe devices expose all configuration, statistics, and debugging registers via the BAR0 MMIO region. The datasheet lists all registers as offsets in this configuration space in Section 9 [20]. We use `ixgbe_type.h` from Intel's driver as machine-readable version of the datasheet[3], it contains defines for all register names and offsets for bit fields.

4.1 NIC Ring API

NICs expose multiple circular buffers called queues or rings to transfer packets. The simplest setup uses only one receive and one transmit queue. Multiple transmit queues are merged on the NIC, incoming traffic is split according to filters or a hashing algorithm if multiple receive queues are configured.

[1] Not the actual ring buffers to prevent user-space applications from crashing the kernel with invalid pointers.

[2] Forwarding 10 Gbit/s with minimum-sized packets on a single 3.0 GHz CPU core leaves a budget of 200 cycles/packet.

[3] This is technically a violation of both our goals about dependencies and lines of code, but we only effectively use less than 100 lines that are just defines and simple structs. There is nothing to be gained from manually copying offsets and names from the datasheet or this file.

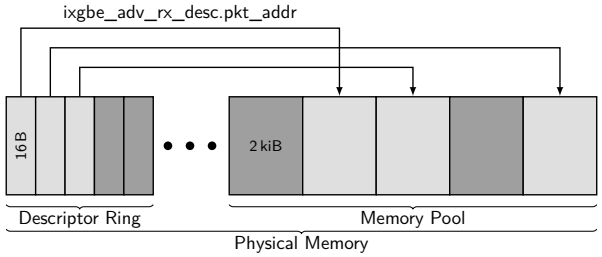

Figure 3: DMA descriptors pointing into a memory pool, note that the packets in the memory are unordered as they can be free'd at different times.

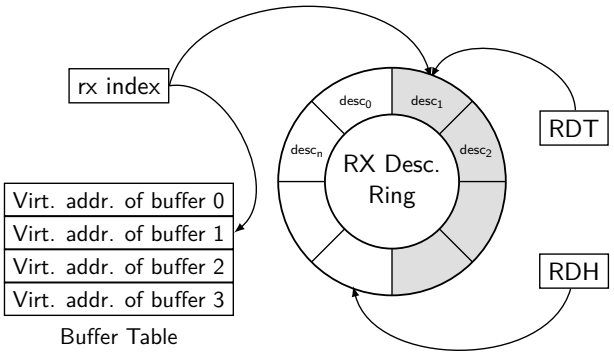

Figure 4: Overview of a receive queue. The ring uses physical addresses and is shared with the NIC.

Both receive and transmit rings work in a similar way: the driver programs a physical base address and the size of the ring. It then fills the memory area with *DMA descriptors*, i.e., pointers to physical addresses where the packet data is stored with some metadata. Sending and receiving packets is done by passing ownership of the DMA descriptors between driver and hardware via a head and a tail pointer. The driver controls the tail, the hardware the head. Both pointers are stored in device registers accessible via MMIO.

The initialization code is in `ixgbe.c` starting from line 114 for receive queues and from line 173 for transmit queues. Further details are in the datasheet in Section 7.1.9 and in the datasheet sections mentioned in the code.

4.1.1 Receiving Packets. The driver fills up the ring buffer with physical pointers to packet buffers in `start_rx_queue()` on startup. Each time a packet is received, the corresponding buffer is returned to the application and we allocate a new packet buffer and store its physical address in the DMA descriptor and reset the ready flag. We also need a way to translate the physical addresses in the DMA descriptor found in the ring back to its virtual counterpart on packet reception. This is done by keeping a second copy of the ring populated with virtual instead of physical addresses, this is then used as a lookup table for the translation.

Figure 3 illustrates the memory layout: the DMA descriptors in the ring to the left contain physical pointers to packet buffers stored in a separate location in a memory pool. The packet buffers in the memory pool contain their physical address in a metadata field. Figure 4 shows the RDH (head) and RDT (tail) registers controlling the ring buffer on the right side, and the local copy containing the virtual addresses to translate the physical addresses in the descriptors in the ring back for the application. `ixgbe_rx_batch()` in `ixgbe.c` implements the receive logic as described by Sections 1.8.2 and 7.1 of the datasheet. It operates on batches of packets to increase performance. A naïve way to check if packets have been received is reading the head register from the NIC incurring a PCIe round trip. The hardware also sets a flag in the descriptor via DMA which is far cheaper to read as the DMA write is handled by the last-level cache on modern CPUs. This is effectively the difference between an LLC cache miss and hit for every received packet.

4.1.2 Transmitting Packets. Transmitting packets follows the same concept and API as receiving them, but the function is more complicated because the interface between NIC and driver is asynchronous. Placing a packet into the ring does not immediately transfer it and blocking to wait for the transfer is infeasible. Hence, the `ixgbe_tx_batch()` function in `ixgbe.c` consists of two parts: freeing packets from previous calls that were sent out by the NIC followed by placing the current packets into the ring. The first part is often called cleaning and works similar to receiving packets: the driver checks a flag that is set by the hardware after the packet associated with the descriptor is sent out. Sent packet buffers can then be free'd, making space in the ring. Afterwards, the pointers of the packets to be sent are stored in the DMA descriptors and the tail pointer is updated accordingly.

Checking for transmitted packets can be a bottleneck due to cache thrashing as both the device and driver access the same memory locations [20]. The 82599 hardware implements two methods to combat this: marking transmitted packets in memory occurs either automatically in configurable batches on device side, this can also avoid unnecessary PCIe transfers. We tried different configurations (code in `init_tx()`) and found that the defaults from Intel's driver work best. The NIC can also write its current position in the transmit ring back to memory periodically (called head pointer write back) as explained in Section 7.2.3.5.2 of the datasheet. However, no other driver implements this feature despite the datasheet referring to the normal marking mechanism as "legacy". We implemented support for head pointer write back on a branch [31] but found no measurable performance improvements or effects on cache contention.

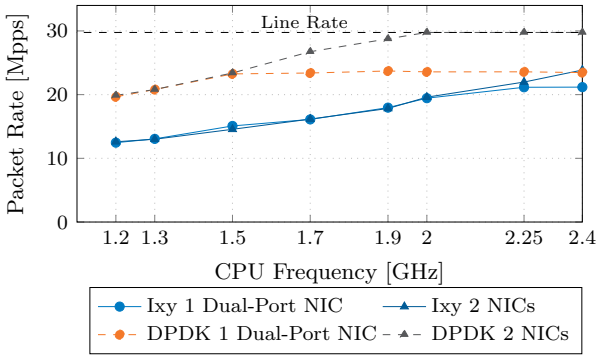

Figure 5: Bidirectional single-core forwarding performance with varying CPU speed, batch size 32.

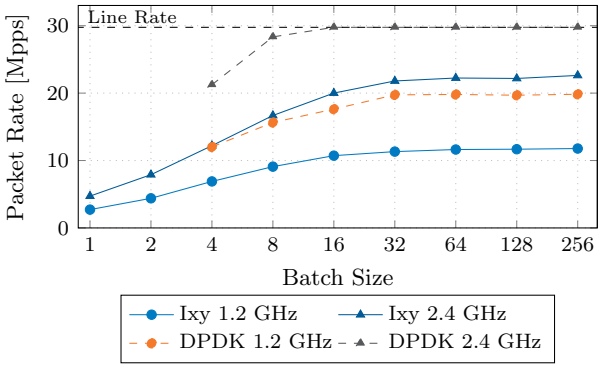

Figure 6: Bidirectional single-core forwarding performance with varying batch size.

4.1.3 Batching. Each successful transmit or receive operation involves an update to the NIC's tail pointer register (RDT or TDT for receive/transmit), a slow operation. This is one of the reasons why batching is so important for performance. Both the receive and transmit function are batched in ixy, updating the register only once per batch.

4.1.4 Offloading Features. ixy currently only enables CRC checksum offloading. Unfortunately, packet IO frameworks (e.g., netmap) are often restricted to this bare minimum of offloading features. DPDK is the exception here as it supports almost all offloading features offered by the hardware. However, its receive and transmit functions pay the price for these features in the form of complexity.

We will try to find a balance and showcase selected simple offloading features in ixy in the future. These offloading features can be implemented in the receive and transmit functions, see comments in the code. This is simple for some features like VLAN tag offloading and more involved for more complex features requiring an additional descriptor containing metadata information.

5 PERFORMANCE EVALUATION

We run the `ixy-fwd` example under a full bidirectional load of 29.76 million packets per second (Mpps), line rate with minimum-sized packets at 2x 10 Gbit/s, and compare it to a custom DPDK forwarder implementing the same features. Both forwarders modify a byte in the packet to ensure that the packet data is fetched into the L1 cache to simulate a somewhat realistic workload.

5.1 Throughput

To quantify the baseline performance and identify bottlenecks, we run the forwarding example while increasing the CPU's clock frequency from 1.2 GHz to 2.4 GHz. Figure 5 compares the throughput of our forwarder on ixy and on DPDK when forwarding across the two ports of a dual-port NIC and when using two separate NICs. The better performance of both ixy and DPDK when using two separate NICs over one dual-port NIC indicates a hardware limit (likely at the PCIe level). We run this test on Intel X520 (82599-based) and Intel X540 NICs with identical results. ixy requires 96 CPU cycles to forward a packet, DPDK only 61. The high performance of DPDK can be attributed to its *vector transmit path* utilizing SIMD instructions to handle batches even better than ixy. This transmit path of DPDK is only used if no offloading features are enabled at device configuration time, i.e., it offers a similar feature set to ixy. Disabling the vector TX path in the DPDK configuration, or using an older version of DPDK, increases the CPU cycles per packet to 91 cycles packet, still slightly faster than ixy despite doing more (checking for more offloading flags). Overall, we consider ixy fast enough for our purposes. For comparison, performance evaluations of older (2015) versions of DPDK, PF_RING, and netmap and required ≈100 cycles/packet for DPDK and PF_RING and ≈120 cycles/packet for netmap [12].

5.2 Batching

Batching is one of the main drivers for performance. DPDK even requires a minimum batch size of 4 when using the SIMD transmit path. Receiving or sending a packet involves an access to the queue index registers, invoking a costly PCIe round-trip. Figure 6 shows how the performance increases as the batch size is increased in the bidirectional forwarding scenario with two NICs. Increasing batch sizes have diminishing returns: this is especially visible when the CPU is only clocked at 1.2 GHz. Reading the performance counters for all caches shows that the number of L1 cache misses per packet increases as the performance gains drop off. Too large batches thrash the L1 cache, possibly evicting lookup data structures in a real application. Therefore, batch sizes should not be chosen too large. Latency is also impacted by the batch size, but the effect is negligible compared to other buffers (e.g., NIC ring size of 512).

Intr./polling	Load	Median	99th perc.	99.99th perc.	Max
Polling	0.1 Mpps	3.8 μs	4.7 μs	5.8 μs	15.8 μs
Intr., no throttling	0.1 Mpps	7.7 μs	8.4 μs	11.0 μs	76.6 μs
Intr., ITR 10 μs	0.1 Mpps	11.3 μs	11.9 μs	15.3 μs	78.4 μs
Intr., ITR 200 μs	0.1 Mpps	107.4 μs	208.1 μs	240.0 μs	360.0 μs
Polling	0.4 Mpps	3.8 μs	4.6 μs	5.8 μs	16.4 μs
Intr., no throttling	0.4 Mpps	7.3 μs	8.2 μs	10.9 μs	53.9 μs
Intr., ITR 10 μs	0.4 Mpps	9.0 μs	14.0 μs	25.8 μs	86.1 μs
Intr., ITR 200 μs	0.4 Mpps	105.9 μs	204.6 μs	236.7 μs	316.2 μs
Polling	0.8 Mpps	3.8 μs	4.4 μs	5.6 μs	16.8 μs
Intr., no throttling	0.8 Mpps	5.6 μs	8.2 μs	10.8 μs	81.1 μs
Intr., ITR 10 μs	0.8 Mpps	9.2 μs	14.1 μs	31.0 μs	70.2 μs
Intr., ITR 200 μs	0.8 Mpps	102.8 μs	198.8 μs	226.1 μs	346.8 μs

Table 1: Forwarding latency by interrupt/poll mode.

App/Function	RX	TX	Forwarding	Memory Mgmt.
ixy-fwd	44.8	14.7	12.3	30.4
ixy-fwd-inline	57.0	28.3	12.5	?*
DPDK l2fwd	35.4	20.4	†6.1	?*
DPDK v1.6 l2fwd‡	41.7	53.7	†6.0	?*

*Memory operations inlined, separate profiling not possible.
†DPDK's driver explicitly prefetches packet data on RX, so this is faster despite performing the same action of changing one byte.
‡Old version 1.6 (2014) of DPDK, far fewer SIMD optimizations, measured on a different system/kernel due to compatibility.

Table 2: Processing time in cycles per packet.

5.3 Interrupts

Interrupts are a common mechanism to reduce power consumption at low loads. However, interrupts are expensive: they require multiple context switches. This makes them unsuitable for high packet rates. NICs commonly feature interrupt throttling (ITR, configured in μs between interrupts on the NIC used here) to prevent overloading the system. Operating systems often disable interrupts periodically and switch to polling during periods of high loads (e.g., Linux NAPI). Our forwarder loses packets in interrupt mode at rates of above around 1.5 Mpps even with aggressive throttling configured on the NIC. All other tests except this one are therefore conducted in pure polling mode.

A common misconception is that interrupts reduce latency, but they actually increase latency. The reason is that an interrupt first needs to wake the system from sleep (sleep states down to C6 are enabled on the test system), trigger a context switch into interrupt context, trigger another switch to the driver and then poll the packets from the NIC[4]. Permanently polling for new packets in a busy-wait loop avoids this at the cost of power consumption.

Table 1 shows latencies at low rates (where interrupts are effective) with and without interrupt throttling (ITR) and polling. Especially tail latencies are affected by using interrupts instead of polling. All timestamps were acquired with a fiber-optic splitter and a dedicated timestamping device taking timestamps of every single packet.

These results show that interrupts with a low throttle rate are feasible at low packet rates. Interrupts are poorly supported in other user space drivers: Snabb offers no interrupts, DPDK has limited support for interrupts (only some drivers) without built-in automatic switching between different modes. Frameworks relying on a kernel driver can use the default driver's interrupt features, especially netmap offers good support for power-saving via interrupts.

[4]The same is true for Linux kernel drivers, the actual packet reception is not done in the hardware interrupt context but in a software interrupt

5.4 Profiling

We run `perf` on `ixy-fwd` running under full bidirectional load at 1.2 GHz with two different NICs using the default batch size of 32 to ensure that the CPU is the only bottleneck. `perf` allows profiling with the minimum possible effect on the performance: throughput drops by only ≈5% while `perf` is running. Table 2 shows where CPU time is spent on average per forwarded packet and compares it to DPDK. Receiving is slower because the receive logic performs the initial fetch, the following functions operate on the L1 cache. ixy's receive function still leaves room for improvements, it is less optimized than the transmit function. There are several places in the receive function where DPDK avoids memory accesses by batching compared to ixy. However, these optimizations were not applied for simplicity in ixy: DPDK's receive function is quite complex and full of SIMD intrinsics leading to poor readability. We also compare an old version of DPDK in the table that did not yet contain as many optimizations; ixy outperforms the old DPDK version at low CPU speeds, but the old DPDK version is ≈10% faster than ixy at higher CPU speeds indicating better utilization of the CPU pipeline.

Overhead for memory management is significant (but still low compared to the 100 cycles/packet we measured in the Linux kernel). 59% of the time is spent in non-batched memory operations and none of the calls are inlined. Inlining these functions increases throughput by 6.5% but takes away our ability to account time spent in them. Overall, the overhead of memory management is larger than we initially expected, but we still think explicit memory management for the sake of a usable API is a good trade-off. This is especially true for ixy aiming at simplicity, but also for other frameworks targeting complex applications. Simple forwarding can easily be done on an exposed ring interface, but anything more complex that does not sent out packets immediately (e.g., because they are processed further on a different core) requires memory management in the user's application with a similar performance impact.

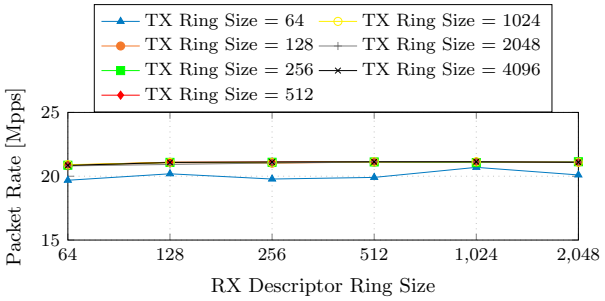

Figure 7: Throughput with varying descriptor ring sizes at 2.4 GHz.

5.5 Queue Sizes

Our driver supports descriptor ring sizes in power-of-two increments between 64 and 4096, the hardware supports more sizes but the restriction to powers of two simplifies wrap-around handling. Linux defaults to a ring size of 256 for this NIC, DPDK's example applications configure different sizes; the `l2fwd` forwarder sets 128/512 RX/TX descriptors. Larger ring sizes such as 8192 are sometimes recommended to increase performance [1] (source refers to the size as kB when it is actually number of packets). Figure 7 shows the throughput of ixy with various ring size combinations. There is no measurable impact on the maximum throughput for ring sizes larger than 64. Scenarios where a larger ring size can still be beneficial might exist: for example, an application producing a large burst of packets significantly faster than the NIC can handle for a very short time.

The second performance factor that is impacted by ring sizes is the overall latency caused by unnecessary buffering. Table 3 shows the latency (measured with MoonGen hardware timestamping [9]) of the ixy forwarder with different ring sizes. The results show a linear dependency between ring size and latency when the system is overloaded, but the effect under lower loads are negligible. Full or near full buffers are no exception on systems forwarding Internet traffic due to protocols like TCP that try to fill up buffers completely [13]. We conclude that tuning tips like setting ring sizes to 8192 [1] are detrimental for latency and likely do not help with throughput. ixy uses a default ring size of 512 at the moment as a trade-off between providing some buffer and avoiding high worst-case latencies.

Ring Sizes	Load	Median	99th perc.	99.9th perc.
64	15 Mpps	5.2 μs	6.4 μs	7.2 μs
512	15 Mpps	5.2 μs	6.5 μs	7.5 μs
4096	15 Mpps	5.4 μs	6.8 μs	8.7 μs
64	*29 Mpps	8.3 μs	9.1 μs	10.6 μs
512	*29 Mpps	50.9 μs	52.3 μs	54.3 μs
4096	*29 Mpps	424.7 μs	433.0 μs	442.1 μs

*Device under test overloaded, packets were lost

Table 3: Forwarding latency by ring size and load.

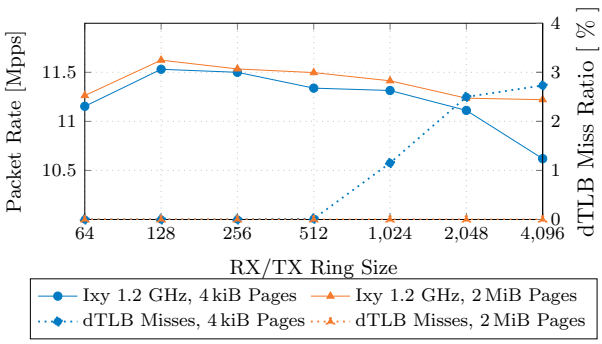

Figure 8: Single-core forwarding performance with and without huge pages and their effect on the TLB.

5.6 Page Sizes without IOMMU

It is not possible to allocate DMA memory on small pages from user space in Linux in a reliable manner without using the IOMMU as described in Section 3.3.2. Despite this, we have implemented an allocator that performs a brute-force search for physically contiguous normal-sized pages from user space. We run this code on a system without NUMA and with transparent huge pages and page-merging disabled to avoid unexpected page migrations. The code for these benchmarks is hidden on a branch [30] due to its unsafe nature on some systems (we did lose a file system to rogue DMA writes on a misconfigured server). Benchmarks varying the page size are interesting despite these problems: kernel drivers and user space packet IO frameworks using kernel drivers only support normal-sized pages. Existing performance claims about huge pages in drivers are vague and unsubstantiated [21, 42].

Figure 8 shows that the impact on performance of huge pages in the driver is small. The performance difference is 5.5% with the maximum ring size, more realistic ring sizes only differ by 1-3%. This is not entirely unexpected: the largest queue size of 4096 entries is only 16 kiB large, storing pointers to up to 16 MiB packet buffers. Huge pages are designed for, and usually used with, large data structures, e.g., big lookup tables for forwarding. The effect measured here is likely larger when a real forwarding application puts additional pressure on the TLB (4096 entries on the CPU used here) due to its other internal data structures. One should still use huge pages for other data structures in a packet processing application, but a driver not supporting them is not as bad as one might expect when reading claims about their importance from authors of drivers supporting them.

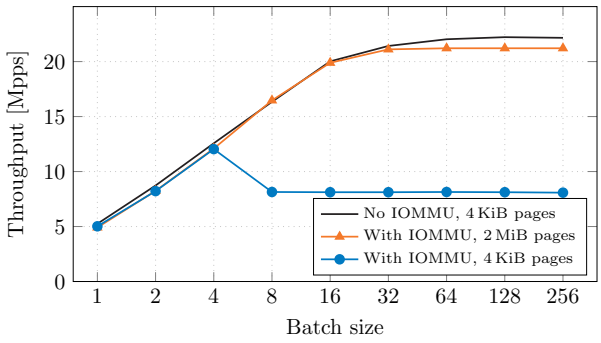

Figure 9: IOMMU impact on single-core forwarding at 2.4 GHz.

Ingress*	Egress*	CPU†	Memory‡	Throughput
Node 0	Node 0	Node 0	Node 0	10.8 Mpps
Node 0	Node 0	Node 0	Node 1	10.8 Mpps
Node 0	Node 0	Node 1	Node 0	7.6 Mpps
Node 0	Node 0	Node 1	Node 1	6.6 Mpps
Node 0	Node 1	Node 0	Node 0	7.9 Mpps
Node 0	Node 1	Node 0	Node 1	10.0 Mpps
Node 0	Node 1	Node 1	Node 0	8.6 Mpps
Node 0	Node 1	Node 1	Node 1	8.1 Mpps

*NUMA node connected to the NIC
†Thread pinned to this NUMA node
‡Memory pinned to this NUMA node

Table 4: Unidirectional forwarding on a NUMA system, both CPUs at 1.2 GHz.

5.7 Page Sizes and IOMMU Overhead

Memory access overhead changes if the device has to go through the IOMMU for every access. Documentation for Intel's IOMMU is sparse: The TLB size is not documented and there are no dedicated performance counters. Neugebauer et al. experimentally determined a TLB size of 64 entries with their pcie-bench framework [33] (vs. 4096 entries in the normal TLB). They note a performance impact for small DMA transactions with large window sizes: 64 byte read throughput drops by up to 70%, write throughput by up to 55%. 256 byte reads are 30% slower, only 512 byte and larger transactions are unaffected [33]. Their results are consistent across four different Intel microarchitectures including the CPU we are using here. They explicitly disable huge pages for their IOMMU benchmark.

Our benchmark triggers a similar scenario when not used with huge pages: We ask the NIC to transfer a large number of small packets via DMA. Note that packets in a batch are not necessarily contiguous in memory: Buffers are not necessarily allocated sequentially and each DMA buffer is 2 kiB large by default, of which only the first n bytes will be transferred. This means only two packets share a 4 kiB page, even if the packets are small. 2 kiB is a common default in other drivers as it allows handling normal sized frames without chaining several buffers (the NIC only supports DMA buffers that are a multiple of 1 kiB). The NIC therefore has to perform several small DMA transactions, i.e., the scenario is equivalent to a large transfer window in pcie-bench.

Figure 9 shows that the IOMMU does not affect the performance if used with 2 MiB pages. However, the default 4 KiB pages (that are safe and easy to use with `vfio` and the IOMMU) are affected by the small TLB in the IOMMU. The impact of the IOMMU on our real application is slightly smaller than in the synthetic pcie-bench tests: The IOMMU costs 62% performance for the commonly used batch size of 32 with small packets when not using huge pages. Running the test with 128 byte packets causes 33% performance loss, 256 byte packets yield identical performance.

However, enabling huge pages completely mitigates the impact of the small TLB in the IOMMU. Note that huge pages for the IOMMU are only supported since the Intel Ivy Bridge CPU generation.

5.8 NUMA Considerations

Non-uniform memory access (NUMA) architectures found on multi-CPU servers present additional challenges. Modern systems integrate cache, memory controller, and PCIe root complex in the CPU itself instead of using a separate IO hub. This means that a PCIe device is attached to only one CPU in a multi-CPU system, access from or to other CPUs needs to pass over the CPU interconnect (QPI on our system). At the same time, the tight integration of these components allows the PCIe controller to transparently write DMA data into the cache instead of main memory. This works even when DCA (direct cache access) is not used (DCA is only supported by the kernel driver, none of the full user space drivers implement it). Intel DDIO (Data Direct I/O) is another optimization to prevent memory accesses by DMA [18]. However, we found by reading performance counters that even CPUs not supporting DDIO do not perform memory accesses in a typical packet forwarding scenario. DDIO is poorly documented and exposes no performance counters, its exact effect on modern systems is unclear. All recent (since 2012) CPUs supporting multi-CPU systems also support DDIO. Our NUMA benchmarks where obtained on a different system than the previous results because we want to avoid potential problems with NUMA for the other setups.

Our test system has one dual-port NIC attached to NUMA node 0 and a second to NUMA node 1. Both the forwarding process and the memory used for the DMA descriptors and packet buffers can be explicitly pinned to a NUMA node. This gives us 8 possible scenarios for unidirectional packet forwarding by varying the packet path and pinning. Table 4 shows the throughput at 1.2 GHz. Forwarding from and to a

NIC at the same node shows one unexpected result: pinning memory, but not the process itself, to the wrong NUMA node does not reduce performance. The explanation for this is that the DMA transfer is still handled by the correct NUMA node to which the NIC is attached, the CPU then caches this data while informing the other node. However, the CPU at the other node never accesses this data and there is hence no performance penalty. Forwarding between two different nodes is fastest when the the memory is pinned to the egress nodes and CPU to the ingress node and slowest when both are pinned to the ingress node. Real forwarding applications often cannot know the destination of packets at the time they are received, the best guess is therefore to pin the thread to the node local to the ingress NIC and distribute packet buffer across the nodes. Latency was also impacted by poor NUMA mapping, we measured an additional 1.7 μs when unnecessarily crossing the NUMA boundary.

6 VIRTIO IMPLEMENTATION

All section numbers for the specification refer to version 1.0 of the VirtIO specification [20]. Function names and line numbers referring to our implementation are hyperlinked to the source code on GitHub.

VirtIO defines different types of operational modes for emulated network cards: legacy, modern, and transitional devices. qemu implements all three modes, the default being transitional devices supporting both the legacy and modern interface after feature negotiation. Supporting devices operating only in modern mode would be the simplest implementation in ixy because they work with MMIO. Both legacy and transitional devices require support for PCI IO port resources making the device access different from the ixgbe driver. Modern-only devices are rare because they are relatively new (2016).

We chose to implement the legacy variant as VirtualBox only supports the legacy operation mode. VirtualBox is an important target as it is the only hypervisor supporting VirtIO that is available on all common operating systems. Moreover, it is very well integrated with Vagrant [17] allowing us to offer a self-contained setup to run ixy on any platform [37].

6.1 Device Initialization and Virtqueues

`virtio_legacy_init()` resets and configures a VirtIO device. It negotiates the VirtIO version and features to use. See specification Section 5.1.3 and 5.1.5 for the available feature flags and initialization steps.

VirtIO supports three different types of queues called Virtqueues: receive, transmit, and command queues. The queue sizes are controlled by the device and are fixed to 256 entries for legacy devices. Setup works the same as in the ixgbe driver: DMA memory for shared structures is allocated

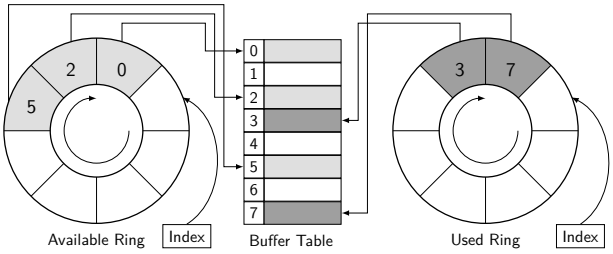

Figure 10: Overview of a Virtqueue. Descriptor table contains physical addresses, the queues indices into the descriptor table.

and passed to the device via a control register. Contrary to queues in ixgbe, a Virtqueue internally consists of a descriptor table and two rings: the *available* and *used* rings. While the table holds the complete descriptors with pointers to the physical addresses and length information of buffers, the rings only contain indices for this table as shown in Figure 10. To supply a device with new buffers, the driver first adds new descriptors into free slots in the descriptor table and then enqueues the slot indices into the *available* ring by advancing its head. Conversely, a device picks up new descriptor indices from this ring, takes ownership of them and then signals completion by enqueuing the indices into the *used* ring, where the driver finalizes the operation by clearing the descriptor from the table. The queue indices are maintained in DMA memory instead of in registers like in the ixgbe implementation. Therefore, the device needs to be informed about all modifications to queues, this is done by writing the queue ID into a control register in the IO port memory region. Our driver also implements batching here to avoid unnecessary updates. This process is the same for sending and receiving packets. Our implementations are in `virtio_legacy_setup_tx/rx_queue()`.

The command queue is a transmit queue that is used to control most features of the device instead of via registers. For example, enabling or disabling promiscuous mode in `virtio_legacy_set_promiscuous()` is done by sending a command packet with the appropriate flags through this queue. See specification Section 5.1.6.5 for details on the command queue. This way of controlling devices is not unique to virtual devices. For example, the Intel XL710 40 Gbit/s configures most features by sending messages to the firmware running on the device [19].

6.2 Packet Handling

Packet transmission in `virtio_tx_batch()` and reception in `virtio_rx_batch()` works similar to the ixgbe driver. The big difference to ixgbe is passing of metadata and offloading information. Virtqueues are not only used for VirtIO network devices, but for other VirtIO devices as well. DMA

descriptors do not contain information specific for network devices. Packets going through Virtqueues have this information prepended in an extra header in the DMA buffer.

This means that the transmit function needs to prepend an additional header to each packet, and our goal to support device-agnostic applications means that the application cannot know about this requirement when allocating memory. ixy handles this by placing this extra header in front of the packet as VirtIO DMA requires no alignment on cache lines. Our packet buffers already contain metadata before the actual packet to track the physical address and the owning memory pool. Packet data starts at an offset of one cache line (64 byte) in the packet buffer, due to alignment requirements of other NICs. This metadata cache line has enough space to accommodate the additional VirtIO header, we have explicitly marked this available area as *head room* for drivers requiring this. Our receive function offsets the address in the DMA descriptor by the appropriate amount to receive the extra header in the head room. The user's ixy application treats the metadata header as opaque data.

6.3 VirtIO Performance

Performance with VirtIO is dominated by the implementation of the virtual device, i.e., the hypervisor, and not the driver in the virtual machine. It is also possible to implement the hypervisor part of VirtIO, i.e., the device, in a separate user space application via the Vhost-user interface of qemu [45]. Implementations of this exist in both Snabb und DPDK. We only present baseline performance measurements running on kvm with Open vSwitch and in VirtualBox, because we are not interested in getting the fastest possible result, but reproducible results in a realistic environment. Optimizations on the hypervisor are out of scope for this paper.

Running ixy in qemu-kvm 2.7.1 on a Xeon E3-1230 V2 CPU clocked at 3.30 GHz yields a performance of only 0.94 Mpps for the `ixy-pktgen` application and 0.36 Mpps for `ixy-fwd`. DPDK is only marginally faster on the same setup: it manages to forward 0.4 Mpps, these slow speeds are not unexpected on unoptimized hypervisors [10]. Performance is limited by packet rate, not data rate. Profiling with 1514 byte packets yield near identical results with a forwarding rate of 4.8 Gbit/s. VMs often send even larger packets with an offloading feature known as generic segmentation offloading offered by VirtIO to achieve higher rates. Profiling on the hypervisor shows that the interconnect is the bottleneck. It fully utilizes one core to forward packets with Open vSwitch 2.6 through the kernel to the second VM. Performance is even worse on VirtualBox 5.2 in our Vagrant setup [37]. It merely achieves 0.05 Mpps on Linux with a 3.3 GHz Xeon E3 CPU and 0.06 Mpps on macOS with a 2.3 GHz Core i7 CPU (606 Mbit/s with 1514 byte packets). DPDK achieves 0.08 Mpps on the macOS setup. Profiling within the VM shows that over 99% of the CPU time is spent on an x86 `OUT` IO instruction to communicate with the virtual device.

7 CONCLUSIONS

We discussed how to build a user space driver for NICs of the ixgbe family which are commonly found in servers and for virtual VirtIO NICs. Our performance evaluation offers some unprecedented looks into performance of user space drivers. ixy allows us to assess effects of individual optimizations, like DMA buffers allocated on huge pages, in isolation. Our driver allowed for a simple port to normal-sized pages without IOMMU, this would be significant change in other frameworks[5]. Not everyone has access to servers with 10 Gbit/s NICs to play around with driver development. However, everyone can build a VM setup to test ixy with our VirtIO driver. Our Vagrant setup is the simplest way to run ixy in a VM, instructions are in our repository [37].

Drivers in High-Level Languages

The implementation presented here is written in C with the hope to be readable by everyone. But there is nothing tying user space drivers to traditional systems programming languages. We also implemented the same driver in Rust, Go, C#, Swift, OCaml, Haskell, Java, JavaScript, and Python to compare these languages for user space drivers [8].

Reproducible Research

Scripts used for the evaluation and our DPDK forwarding application used for comparison are available in [39]. We used commit *df1cddbb* of ixy for the evaluation of ixgbe and virtio, commit *a0f618d* on a branch [30] for the normal sized pages. Most results were obtained on an Intel Xeon E5-2620 v3 2.4 GHz CPU running Debian 9 (kernel 4.9) with a dual port Intel 82599ES NIC. The NUMA results where obtained on a system with two Intel Xeon E5-2630 v4 2.2 GHz CPUs with Intel X550 NICs. Turboboost, Hyper-Threading, and power-saving features were disabled. VirtIO results were obtained on various systems and hypervisors as described in the evaluation section. All loads where generated with MoonGen [9] and its `l2-load-latency.lua` script.

Acknowledgments

This work was supported by the German-French Academy for the Industry of the Future. We would like to thank Simon Ellmann, Masanori Misono, and Boris-Chengbiao Zhou for valuable contributions to ixy and/or this paper.

[5] DPDK offers `--no-huge`, but this setting is incompatible with most DMA drivers due to the aforementioned safety issues.

REFERENCES

[1] Jamie Bainbridge and Jon Maxwell. 2015. Red Hat Enterprise Linux Network Performance Tuning Guide. *Red Hat Documentation* (March 2015). https://access.redhat.com/sites/default/files/attachments/20150325_network_performance_tuning.pdf.

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *ACM/IEEE ANCS*.

[3] Gilberto Bertin. 2015. Single RX queue kernel bypass in Netmap for high packet rate networking. (Oct. 2015). https://blog.cloudflare.com/single-rx-queue-kernel-bypass-with-netmap/.

[4] N. Bonelli, S. Giordano, and G. Procissi. 2016. Network Traffic Processing With PFQ. *IEEE Journal on Selected Areas in Communications* 34, 6 (June 2016), 1819–1833. https://doi.org/10.1109/JSAC.2016.2558998

[5] DPDK Project. 2019. DPDK: Supported NICs. (2019). http://dpdk.org/doc/nics.

[6] DPDK Project. 2019. DPDK User Guide: Overview of Networking Drivers. (2019). http://dpdk.org/doc/guides/nics/overview.html.

[7] DPDK Project. 2019. DPDK Website. (2019). http://dpdk.org/.

[8] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. 2019. The Case for Writing Network Drivers in High-Level Programming Languages. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*.

[9] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan.

[10] Paul Emmerich, Daniel Raumer, Sebastian Gallenmüller, Florian Wohlfart, and Georg Carle. 2017. Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis. *Journal of Network and Systems Management* (July 2017). https://doi.org/10.1007/s10922-017-9417-0

[11] FreeBSD Project. 2017. NETMAP(4). In *FreeBSD Kernel Interfaces Manual*. FreeBSD 11.1-RELEASE.

[12] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. 2015. Comparison of Frameworks for High-Performance Packet IO. In *Architectures for Networking and Communications Systems (ANCS)*. ACM, Oakland, CA, 29–38.

[13] Jim Gettys and Kathleen Nichols. 2011. Bufferbloat: Dark buffers in the internet. *Queue* 9, 11 (2011), 40.

[14] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Netdev 2.1, The Technical Conference on Linux Networking*.

[15] Gorrie, L et al. 2019. Snabb: Simple and fast packet networking. (2019). https://github.com/snabbco/snabb.

[16] Michael Haardt. 1993. ioperm(2). In *Linux Programmer's Manual*.

[17] HashiCorp. 2019. Vagrant website. (2019). https://www.vagrantup.com/.

[18] Intel. 2012. Intel Data Direct I/O Technology (Intel DDIO): A Primer. (2012). https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html.

[19] Intel. 2014. Intel Ethernet Controller XL710 Datasheet Rev. 2.1.

[20] Intel. 2016. Intel 82599 10 GbE Controller Datasheet Rev. 3.3.

[21] Intel. 2019. DPDK Getting Started Guide for Linux. (2019). http://dpdk.org/doc/guides/linux_gsg/sys_reqs.html.

[22] IO Visor Project. 2019. BPF and XDP Features by Kernel Version. (2019). https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp.

[23] IO Visor Project. 2019. Introduction to XDP. (2019). https://www.iovisor.org/technology/xdp.

[24] Jim Thompson. 2017. DPDK, VPP & pfSense 3.0. In *DPDK Summit Userspace*.

[25] Jonathan Corbet. 2017. User-space networking with Snabb. In *LWN.net*.

[26] Michael Kerrisk. 2004. mlock(2). In *Linux Programmer's Manual*.

[27] Linux Foundation. 2017. Networking Industry Leaders Join Forces to Expand New Open Source Community to Drive Development of the DPDK Project. (April 2017). Press release.

[28] Linux Kernel Documentation. 2019. Page migration. (2019). https://www.kernel.org/doc/Documentation/vm/page_migration.

[29] Linux Kernel Documentation. 2019. VFIO - Virtual Function I/O. (2019). https://www.kernel.org/doc/Documentation/vfio.txt.

[30] Maximilian Pudelko. 2019. ixy - DMA allocator on normal-sized pages. (2019). https://github.com/pudelkoM/ixy/tree/contiguous-pages.

[31] Maximilian Pudelko. 2019. ixy - head pointer writeback implementation. (2019). https://github.com/pudelkoM/ixy/tree/head-pointer-writeback.

[32] Robert Morris, Eddie Kohler, John Jannotti, and M Frans Kaashoek. 1999. The Click modular router. In *Operating Systems Review - SIGOPS*, Vol. 33. 217–231.

[33] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *SIGCOMM 2018*. ACM, 327–341.

[34] ntop. 2014. PF_RING ZC (Zero Copy). (2014). http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[35] OASIS VIRTIO TC. 2016. Virtual I/O Device (VIRTIO) Version 1.0. (March 2016). http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf.

[36] Open vSwitch Project. 2019. Open vSwitch with DPDK. (2019). http://docs.openvswitch.org/en/latest/intro/install/dpdk/.

[37] Paul Emmerich. 2019. ixy Vagrant setup. (2019). https://github.com/emmericp/ixy/tree/master/vagrant.

[38] Paul Emmerich et al. 2019. ixy code. (2019). https://github.com/emmericp/ixy.

[39] Paul Emmerich, Simon Bauer. 2019. Scripts used for the performance evaluation. (2019). https://github.com/emmericp/ixy-perf-measurements.

[40] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 117–130.

[41] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O.. In *USENIX Annual Technical Conference*. 101–112.

[42] Snabb Project. 2018. Tuning the performance of the lwaftr. (2018). https://github.com/snabbco/snabb/blob/master/src/program/lwaftr/doc/performance.md.

[43] Snort Project. 2015. Snort 3 User Manual. (2015). https://www.snort.org/downloads/snortplus/snort_manual.pdf.

[44] Solarflare. 2019. OpenOnload Website. (2019). http://www.openonload.org/.

[45] Virtual Open Systems Sarl. 2014. Vhost-user Protocol. (2014). https://github.com/qemu/qemu/blob/stable-2.10/docs/interop/vhost-user.txt.

[46] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 43–56.