User Space Network Drivers

Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle
Technical University of Munich
Department of Informatics
Chair of Network Architectures and Services
{*emmericp*|*pudelko*|*bauersi*|*carle*}*@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 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. Ixy focuses on the bare essentials of user space packet processing: a packet forwarder including the whole NIC driver uses less than 1000 lines of C code.

We discuss how ixy implements 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`.

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 [36] and the Click modular router [33]. Writing kernel code is not only a relatively cumbersome process with slow turn-around 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 [35]. Click was ported to both netmap [40] and DPDK for the same reasons [2]. Other projects also moved kernel-based code to specialized user space code [27, 42].

Developers and researchers still 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 [30]. This means that there is no academic paper describing its architecture or implementation. The netmap paper [40] 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.

We present ixy, a user space packet framework that is architecturally similar to DPDK [7] and Snabb [17]. Both use full user space drivers, unlike netmap [40], 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 setting. Our whole architecture aims at simplicity and is trimmed down to the bare minimum. We currently support the Intel ixgbe family of NICs and virtual VirtIO NICs. A packet forwarding application is less than 1000 lines of code including the whole driver. 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 82599 driver needs 5400 lines of code just to handle receiving and sending packets in a highly optimized way. 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 [8]. Further, we publish all scripts used for our evaluation [10].

The remainder of this paper is structured as follows.

We first discuss background and related work, i.e., the basics of other user space packet IO frameworks and the differences between them. We then look at ixy's design at a high level in Section 3 before diving into implementation (4) and performance (5) of our ixgbe driver. Section 6 discusses our VirtIO driver before concluding with an explanation on reproducing our results and running ixy in a virtual machine in Section 7.

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 and those that re-implement the whole driver in user space.

Examples for the former category are netmap [40], PF_RING ZC [34], pfq [4], OpenOnload [43], and XDP [26]. 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 normal driver here, but the driver is attached to the application directly instead of to the normal kernel datapath. This has the advantage that integrating existing kernel components or forwarding packets to the default network stack is relatively simple 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, offloading features are often poorly supported, e.g., netmap supports no hardware checksums or tunnel en-/decapsulation features at all [13].

In particular, netmap [40] and XDP [26] 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 [16]. However, it is currently not feasible to use as a foundation for more complex applications due to limited functionality and restrictions imposed by running as eBPF code in the kernel. 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 4.15 supports fewer drivers than DPDK [25, 5].

DPDK [7] and Snabb [17] 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. 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 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's 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 abstraction from the driver is similar to DPDK.

3 Design

The language of choice is C as the lowest common denominator of systems programming languages.

Our design goals for ixy 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 [17] has similar design goals; ixy tries to be one order of magnitude simpler. For example, Snabb targets 10,000 lines of code [28], 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've tested our ixgbe driver on Intel X550, X540, and 82599ES (aka X520) 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's based on PCI instead of PCIe, requiring a different approach to implement the driver in user space.

3.3 User Space Drivers in Linux

All function names in the following sections are clickable hyperlinks to our source code on GitHub.

Linux exposes all necessary interfaces to write full user space drivers via 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 configured for the NIC while ixy is running. The only capability that is missing is handling interrupts which could be done by using the `uio_pci_generic` driver for the NIC. Ixy only supports poll-mode at the moment to keep the code simple.

One needs to understand how a driver communicates with a device to understand how a driver can be written in user space. This overview skips details but is sufficient to understand how ixy or similar frameworks work. 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. Linux exposes all BARs in the `sysfs` pseudo filesystem, a privileged process can simply `mmap` them into its address space. Devices commonly expose their configuration registers via this interface where normal reads and writes can be used to access the register. For example, ixgbe NICs expose all configuration, statistics, and debugging registers via the BAR0 address space. The datasheet [22] lists all registers as offsets in this memory area. Our implementation of this mapping can be found in `pci_map_resource()` in `pci.c`.

VirtIO (in the version we are implementing) 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)` [18], 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 also exposes IO port BARs via `sys` 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.

3.3.2 DMA in User Space

DMA is initiated by the PCI device and allows it to read/write arbitrary physical addresses. This is used to read/write packet data and to transfer the DMA descriptors (pointers to packet data and offloading information) 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`. The user space driver hence 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`.

Memory used for DMA transfer must stay resident in physical memory. `mlock(2)` [29] 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 [31]. 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, 41] despite being crucial for reliable allocation of DMA memory. If Linux ever starts moving explicitly allocated huge pages in physical memory, a new memory allocation method is required for all full user space driver frameworks. The `uio` framework with its `uio_pci_generic` driver is one candidate.

3.4 Memory Management

Ixy builds on an API with explicit memory allocation similar to DPDK which is a very different approach from netmap [40] that exposes a replica 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 [40]. Hence, netmap exposes replicas of the ring buffers[1] to the application, and it is then up to the application to handle memory. 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 memory 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 prefills 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.

3.5 Memory Pools and Multi-Threading

Packets may be passed to different threads, for example, a service function chaining application might run different network functions on different CPU cores and pass packets between them. Allocating and freeing packets will happen in different threads in this case as memory management is handled in the receive and transmit functions. Packets must be returned to the memory pool they were allocated from (they keep a reference to the pool) to prevent starving or overflowing pools when forwarding unidirectionally. Therefore, memory pools must be thread-safe for such an application. Memory pools in ixy

[1] Not the actual ring buffers to prevent the user-space application 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.

are currently not thread-safe; they are based on a free list kept in a stack, making bulk operations on the pools trivial and fast. Lock-free stacks or queues could be used, but these data structures are complicated [44]. We do not want the memory pool to be the most complex and hard to understand part of ixy – we therefore do not support passing packets between threads at the moment.

Choosing the right data structure for memory management also affects performance beyond the efficiency of the data structure itself. A stack is better than a queue: it improves temporal cache locality because it recycles packets immediately. DPDK is an interesting case study as they offer thread-safe memory pools. Free buffers are kept in a lock-free queue and each thread keeps a thread-local stack as a cache: an unsynchronized local stack is faster than a lock-free data structure. This works well if multiple threads share a memory pool but do not pass packets between each other. But the cache does not help if the "producers" and "consumers" are separate threads because they exhaust their cache and effectively fall back to the queue. DPDK added a memory pool backed by a stack protected with a spin-lock specifically for such applications because effective cache usage is more important than a lock-free data structure in practice [20].

3.6 Security Considerations

Applications built with ixy require root access to access the hardware, the same is true for virtually all other packet processing frameworks. The only noteworthy exception here is netmap which can grant unprivileged users access and performs checks on user-provided data in the kernel interfaces. Despite the need for root access, the other frameworks are still an improvement over the previous solution: custom kernel modules running C code. The user space solutions can use safer languages, for example, the Snabb drivers are written in Lua[3] [17].

It is possible to drop all privileges using `seccomp(2)` once the PCIe memory regions have been `open`ed or `mmap`ed and all necessary DMA memory has been allocated. We have implemented this in ixy on a branch [39]. However, this is still insecure – the device is under full control of the application and it has full access via DMA to the whole memory. Modern CPUs offer a solution: the IO memory management unit (IOMMU) allows using virtual addresses, translation, and protection for PCIe devices. IOMMUs are available on CPUs offering hardware virtualization features as it was 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" [32] beside virtual machines. This, combined with dropping privileges after initialization (or delegating initialization to a separate process), allows implementing a secure user space driver that requires no privileged access during operation.

[3] However, they make extensive use of memory-unsafe operations with LuaJIT for performance reasons

4 ixgbe Implementation

All line numbers referenced in this Section are for commit `df1cddb` of ixy. All page numbers and section numbers for the Intel datasheet refer to revision 3.3 (March 2016) of the 82599ES datasheet [22]. Function names and line numbers are hyperlinked to the implementation.

ixgbe devices expose all configuration, statistics, and debugging registers via the BAR0 MMIO region. The datasheet [22] lists all registers as offsets in this configuration space. We use `ixgbe_type.h` from Intel's driver as machine-readable version of the datasheet[4], 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. 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 tail pointer. The driver controls the tail, hardware the head. Both pointers are stored in device registers accessible via MMIO.

The initialization code is in `ixgbe.c` starting from line 124 for receive queues and from line 179 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 trans-

[4] This is technically a violation of both our goal 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 copy & pasting offsets and names from the datasheet or this file.

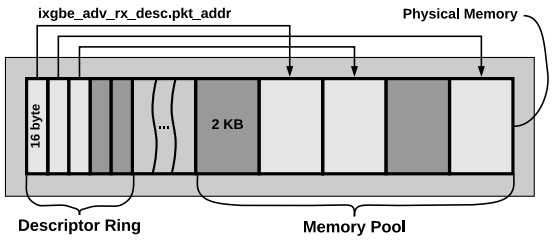

Figure 1: DMA descriptors pointing into a memory pool, note that the packets in the memory are unordered as they can be free'd at different times.

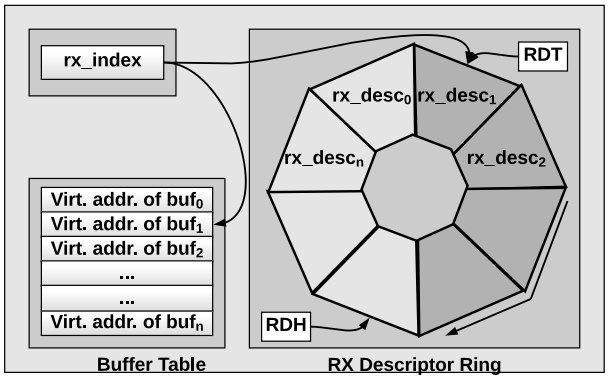

Figure 2: Overview of a receive queue. The ring uses physical addresses and is shared with the NIC.

late 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 1 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 2 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 [22]. 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 [38] but found no measurable performance improvements or effects on cache contention.

4.1.3 Batching

Each successful transmit or receive operation involves an update to the NIC's tail pointer register (RDT and 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, as explained earlier its receive and transmit functions pay the price for these features in the form of complexity.

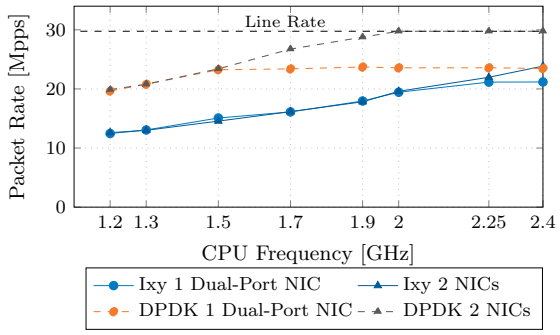

Figure 3: Bidirectional single-core forwarding performance with varying CPU speed, batch size 32.

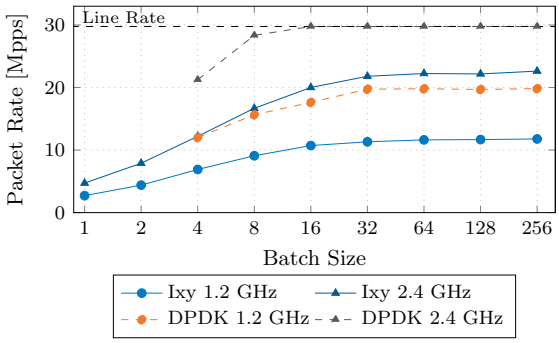

Figure 4: Bidirectional single-core forwarding performance with varying batch size.

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	?*

*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.

Table 1: Processing time in cycles per packet.

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 3 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 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, we have previously studied the performance of older versions of DPDK, PF_RING, and netmap and measured a performance of $\approx$ 100 cycles/packet for DPDK and PF_RING and $\approx$ 120 cycles/packet for netmap [14].

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 4 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 sizes are an order of magnitude larger than the batch size).

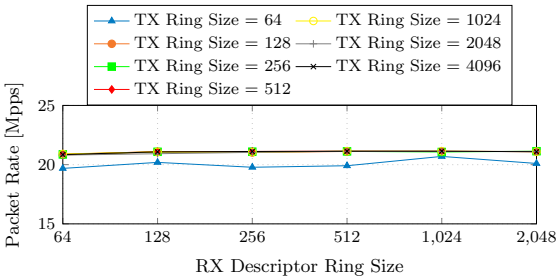

Figure 5: Throughput with varying descriptor ring sizes.

5.3 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 CPU is the only bottleneck. `perf` allows profiling with the minimum possible effect on the performance: throughput drops by only $\approx 5\%$ while `perf` is running. Table 1 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.

Overhead for memory management is significant (but still low compared to 100 cycles/packet 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 worthwhile 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. Moreover, 30 cycles per packet that could be saved is still a tiny improvement compared to other architectural decisions like batch processing that reduces per-packet processing costs by 300 cycles when going from no batching to a batch size of 32.

Ring sizes	Load	Median	99th perc.	99.9th perc.
64	15 Mpps	7.7 µs	8.8 µs	9.6 µs
512	15 Mpps	7.7 µs	8.9 µs	9.9 µs
4096	15 Mpps	7.9 µs	9.2 µs	11.1 µs
64	*29 Mpps	10.7 µs	11.5 µs	13.0 µs
512	*29 Mpps	53.3 µs	54.7 µs	56.7 µs
4096	*29 Mpps	427.1 µs	435.4 µs	444.5 µs

*Device under test overloaded, packets were lost

Table 2: Forwarding latency by ring size and load.

5.4 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 simplify 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 5 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 2 shows the latency (measured with MoonGen hardware timestamping [11]) 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 [15]. We conclude that tuning tipps like setting a ring size 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.

5.5 Page Sizes

It is not possible to allocate DMA memory on small pages from user space in Linux in a reliable manner 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 not available in the main repo but on a

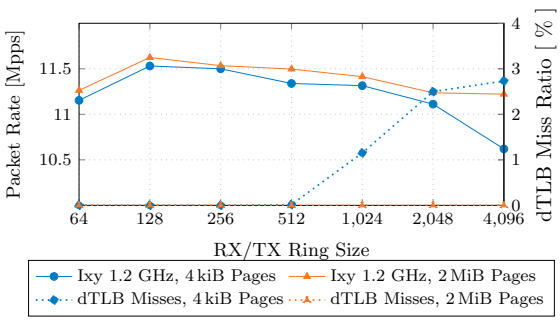

Figure 6: Single-core forwarding performance with and without huge pages and their effect on the TLB.

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 3: Unidirectional forwarding on a NUMA system, CPU at 1.2 GHz

branch [37] due to its unsafe nature on some systems. Benchmarks varying the page size are interesting despite these problems: kernel drivers (and user space packet IO frameworks using them) often only support normal-sized pages. Existing performance claims about huge pages in drivers are vague and unsubstantiated [21, 41].

Figure 6 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 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 (e.g., netmap) is not as bad as one might expect when reading claims about their importance from authors of drivers supporting them.

5.6 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 further optimization to prevent memory accesses by DMA [23]. 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 wanted to avoid potential problems with NUMA for the other setups.

A multi-CPU system consists of multiple NUMA nodes, each has its own CPU, memory, and PCIe devices. 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 3 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 when forwarding between two ports on one NUMA node. Latency comparisons between forwarding within one node vs. for-

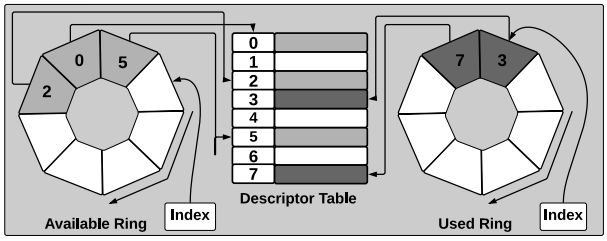

Figure 7: Overview of a Virtqueue. Descriptor table contains physical addresses, the queues indices into the descriptor table.

warding between two nodes where not possible in a fair manner in this system as the NICs use different physical layers with different latencies: the 10GBASE-T NIC has more than 2 μs additional latency.

6 VirtIO Implementation

All line numbers referenced in this Section are for commit `df1cddb` of ixy. All section numbers for the specification refer to version 1.0 of the VirtIO specification [22]. Function names are hyperlinked to the implementation on GitHub containing further references to the relevant specification sections.

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 because VirtualBox only supports the legacy operation mode. VirtualBox is an important target for ixy as it is the only hypervisor supporting VirtIO that is available on all common operating systems. Moreover, it is very well integrated with Vagrant [19] allowing us to offer a full self-contained setup to run ixy on any platform [9].

6.1 Device Initialization and Virtqueues

`virtio_legacy_init()` resets and configures a VirtIO device. It negotiates the VirtIO version and features to use, we do not try to negotiate any advanced features but the support for checksum-free transfer of packets between VMs. 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 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 7. 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 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 [24].

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. Therefore, the DMA descriptor does 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 applica-

tion 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 qemu with Open vSwitch and in VirtualBox, because we are not interested in getting the fastest possible result, but results in an environment that we expect our users to have. Optimizations on the device side are out of scope for this paper.

Running ixy in qemu 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 [12]. 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 [9]. 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/hypervisor.

7 Conclusions: Reproducible Research

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 goal is not build yet another packet IO framework – but a tool for education. Therefore, reproducible research is important to us.

The full code of ixy and the scripts used to to reproduce these results is available on GitHub [8, 10]. Our DPDK forwarding application used for comparison is available in [10]. We used commit `df1cddbb` of ixy for the evaluation of ixgbe and virtio, commit `a0f618d` on a branch [37] for the normal sized pages. Most results were obtained on an Intel Xeon E5-2620 v3 2.4 GHz CPU running Debian 9.3 (kernel 4.9) with a dual port Intel X520-T2 (82599ES) NIC and a dual port X540-T2 NIC. The NUMA results where obtained on a system with two Intel Xeon E5-2630 v4 2.2 GHz CPUs with the same NICs and operating system. 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 [11] and its `l2-load-latency.lua` script.

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, this would be significant change in other frameworks[5]. Not everyone has access to servers with 10 Gbit/s NICs to reproduce these results. 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 on any operating system in VirtualBox, instructions are in our repository [9]. VirtualBox turned out to be slower by a factor of 20 than a setup on `qemu-kvm` which is also relatively easy to build. We have validated Ixy's functionality on a Proxmox 4.4 hypervisor and with `virsh/libvirt` on Ubuntu 16.04.

References

[1] BAINBRIDGE, J., AND MAXWELL, J. Red Hat Enterprise Linux Network Performance Tuning Guide. *Red Hat Documentation* (Mar. 2015). Available at `https://access.redhat.com/sites/default/files/attachments/20150325_network_performance_tuning.pdf`.

[2] BARBETTE, T., SOLDANI, C., AND MATHY, L. Fast userspace packet processing. In *ACM/IEEE ANCS* (2015).

[3] BERTIN, G. 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/`.

[5] DPDK offers `--no-huge`, but this setting is incompatible with DMA drivers

[4] BONELLI, N., GIORDANO, S., AND PROCISSI, G. Network traffic processing with pfq. *IEEE Journal on Selected Areas in Communications 34*, 6 (June 2016), 1819–1833.

[5] DPDK PROJECT. DPDK: Supported NICs. `http://dpdk.org/doc/nics`. Last visited 2018-02-01.

[6] DPDK PROJECT. DPDK User Guide: Overview of Networking Drivers. `http://dpdk.org/doc/guides/nics/overview.html`. Last visited 2018-02-01.

[7] DPDK PROJECT. DPDK Website. `http://dpdk.org/`. Last visited 2018-02-01.

[8] EMMERICH, P. ixy code. `https://github.com/emmericp/ixy`.

[9] EMMERICH, P. ixy Vagrant setup. `https://github.com/emmericp/ixy/tree/master/vagrant`.

[10] EMMERICH, P. Scripts used for the performance evaluation. `https://github.com/emmericp/ixy-perf-measurements/tree/full-paper`.

[11] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F., AND CARLE, G. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)* (Tokyo, Japan, Oct. 2015).

[12] EMMERICH, P., RAUMER, D., GALLENMÜLLER, S., WOHLFART, F., AND CARLE, G. Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis. *Journal of Network and Systems Management* (July 2017).

[13] FREEBSD PROJECT. NETMAP(4). In *FreeBSD Kernel Interfaces Manual* (2017), FreeBSD 11.1-RELEASE.

[14] GALLENMLLER, S., EMMERICH, P., WOHLFART, F., RAUMER, D., AND CARLE, G. Comparison of Frameworks for High-Performance Packet IO. In *Architectures for Networking and Communications Systems (ANCS)* (Oakland, CA, 2015), ACM, pp. 29–38.

[15] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark buffers in the internet. *Queue 9*, 11 (2011), 40.

[16] GILBERTO BERTIN. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Netdev 2.1, The Technical Conference on Linux Networking* (May 2017).

[17] GORRIE, L ET AL. Snabb: Simple and fast packet networking. `https://github.com/snabbco/snabb`.

[18] HAARDT, M. ioperm(2). In *Linux Programmer's Manual* (1993).

[19] HASICORP. Vagrant website. `https://www.vagrantup.com/`. Last visited 2018-02-02.

[20] HUNT, D. mempool: add stack (lifo) mempool handler, 2016. Mailing list post. `http://dpdk.org/ml/archives/dev/2016-July/043106.html`.

[21] INTEL. DPDK Getting Started Guide for Linux. `http://dpdk.org/doc/guides/linux_gsg/sys_reqs.html`. Last visited 2018-02-01.

[22] Intel 82599 10 GbE Controller Datasheet Rev. 3.3. Intel.

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

[24] Intel Ethernet Controller XL710 Datasheet Rev. 2.1. Intel.

[25] IO VISOR PROJECT. BPF and XDP Features by Kernel Version. `https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md#xdp`. Last visited 2018-02-01.

[26] IO VISOR PROJECT. Introduction to XDP. `https://www.iovisor.org/technology/xdp` Last visited 2018-02-01.

[27] JIM THOMPSON. DPDK, VPP & pfSense 3.0. In *DPDK Summit Userspace* (Sept. 2017).

[28] JONATHAN CORBET. User-space networking with Snabb. In *LWN.net* (Feb. 2017).

[29] KERRISK, M. mlock(2). In *Linux Programmer's Manual* (2004).

[30] LINUX FOUNDATION. Networking Industry Leaders Join Forces to Expand New Open Source Community to Drive Development of the DPDK Project, Apr. 2017. Press release.

[31] LINUX KERNEL DOCUMENTATION. Page migration. `https://www.kernel.org/doc/Documentation/vm/page_migration`.

[32] LINUX KERNEL DOCUMENTATION. VFIO - Virtual Function I/O. `https://www.kernel.org/doc/Documentation/vfio.txt`.

[33] MORRIS, R., KOHLER, E., JANNOTTI, J., AND FRANS KAASHOEK, M. The click modular router. In *Operating Systems Review - SIGOPS* (Dec. 1999), vol. 33, pp. 217–231.

[34] NTOP. PF_RING ZC (Zero Copy). `http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/`. Last visited 2017-11-30.

[35] OPEN VSWITCH PROJECT. Open vSwitch with DPDK. `http://docs.openvswitch.org/en/latest/intro/install/dpdk/` Last visited 2018-02-01.

[36] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 117–130.

[37] PUDELKO, M. ixy - DMA allocator on normal-sized pages. `https://github.com/pudelkoM/ixy/tree/contiguous-pages`.

[38] PUDELKO, M. ixy - head pointer writeback implementation. `https://github.com/pudelkoM/ixy/tree/head-pointer-writeback`.

[39] PUDELKO, M. ixy - seccomp implementation. `https://github.com/pudelkoM/ixy/tree/seccomp`.

[40] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference* (2012), pp. 101–112.

[41] SNABB PROJECT. Tuning the performance of the lwaftr. `https://github.com/snabbco/snabb/blob/master/src/program/lwaftr/doc/performance.md`. Last visited 2018-02-01.

[42] SNORT PROJECT. Snort 3 User Manual. `https://www.snort.org/downloads/snortplus/snort_manual.pdf` Last visited 2018-02-01.

[43] SOLARFLARE. OpenOnload Website. `http://www.openonload.org/`. Last visited 2017-11-30.

[44] SUTTER, H. Lock-Free Code: A False Sense of Security. *Dr. Dobb's Journal* (Sept. 2008).

[45] VIRTUAL OPEN SYSTEMS SARL. Vhost-user Protocol, 2014. `https://github.com/qemu/qemu/blob/stable-2.10/docs/interop/vhost-user.txt`.

[46] YASUKATA, K., HONDA, M., SANTRY, D., AND EGGERT, L. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 43–56.