1. Introduction

The networking world is moving from hardware network functions to software ones to gain flexibility. This brings new problems to light in the network stacks of mainstream operating systems, which were not designed for this use case. In response to this move, the kernel-bypass model for software networking appeared, designed for low latency and high throughput. However, one area of the stack that remains under-explored is network drivers. We present the state of network functions, stacks and drivers in Section 2.

Modern network cards contain powerful and complex hardware offloads, but their core features are conceptually simple. Network cards fetch requests and return responses to software using data structures named descriptors. The main complexity for packet reception and transmission is the descriptor ownership mechanism. We present the basics of modern network cards in Section 3.

The current network driver model is too flexible for the needs of common network functions, which must pay the complexity costs of modern drivers without reaping their benefits. This is mainly because the current driver model allows network functions to process packets out of order, a powerful feature that is not needed in many of the core functions making up the Internet's backbone. We formalize the current driver model for network cards and propose our conceptually simplified version in Section 4.

We implement our new driver model for the Intel 82599, a modern 10 Gb/s Ethernet controller. Our implementation uses the model's insights and stays as simple as possible: it is only 550 lines of C code. Its key features are a minimal number of operations thanks to the driver design and to modern network card features, as well as some simple but powerful scheduling algorithms. We present our driver, which we call "TinyNF", in Section 5.

This paper's core hypotheses are that our simpler model (1) makes network functions easier to formally verify, (2) is faster than the current most complex driver model that can be formally verified, (3) provides competitive performance against the fastest state-of-the-art drivers regardless of complexity, and (4) is applicable to most network functions that are deployed today.

We show that hypotheses (1) and (2) hold in Section 6. Our driver has exponentially fewer code paths than current drivers and can thus be used to formally verify network functions in 7x less time than with a state-of-the-art driver while offering 2.5x the throughput, as well as lower median and tail latency.

We show that hypothesis (3) holds in Section 7, with the surprising observation that our driver outperforms the state of the art using real network functions even though it loses on a synthetic "no-op" function. This is because our driver slows down less when running real functions due to having room to grow in instruction-level parallelism and cache use.

We provide evidence for hypothesis (4) in Section 8, showing that our model is applicable to most of the low-level network infrastructure, either running on bare metal or as virtualized network functions.

We believe that the separation in common use between "drivers" and other software is blurry, and we argue that it hinders progress. This situation is worse in networking due to the lack of good baselines for benchmarks, leading to driver optimizations that increase complexity but may not increase performance in the real world. Our minimal driver also highlights opportunities in hardware documentation. We discuss these issues in Section 9.

In summary, we make the following contributions: (1) a simplified driver model for network functions that process packets in order, (2) a formally verified driver based on our model that is easier to reason about and faster on realistic workloads than existing drivers, and (3) evidence that the current standard of benchmarking for network drivers leads to suboptimal performance in the real world.

2. Background on network functions

In this section, we introduce network functions: packet-processing appliances performing tasks such as routing, rate limiting, access control or caching.

Hardware network functions are the traditional way to implement network functions for high-traffic networks. They are physical boxes with custom hardware that are part of the network, distinct from standard computers.

They are typically robust because fixing hardware bugs after deployment is not possible, thus engineers must test them extensively before deployment. However, they are not flexible because they cannot be modified after deployment. Changing a network's policies can require replacing the hardware entirely.

Software network functions run on general-purpose hardware such as x86 and mainstream operating systems such as Linux, communicating with network cards through a software stack that includes drivers and implementations of protocols such as IP and TCP.

The networking world is moving to software network functions to increase flexibility. Software network functions are flexible since they have low deployment costs. This means correctness guarantees, while important, are not a hard requirement for deployment.

Verifying the correctness of software network functions is an open problem, with recent work showing it is easier than the intractable problem of general software verification. The Vigor [33] project verifies network functions without human interaction, but cannot deal with common optimizations such as parallelism or batch processing.

The other key concern of software network function is performance. This includes variability, since worst-case performance determines the guarantees network operators can offer. These guarantees turn into business concerns such as Service Level Agreements.

To illustrate how crucial performance is, consider the time budget for processing a 64-byte packet and its 20-byte Ethernet header at 10 Gb/s: $(64+20) * 8 / (10*10^9) = 67.2$ns. This is the same order of magnitude as a memory read; a network function will exceed its time budget if it needs data outside the CPU cache.

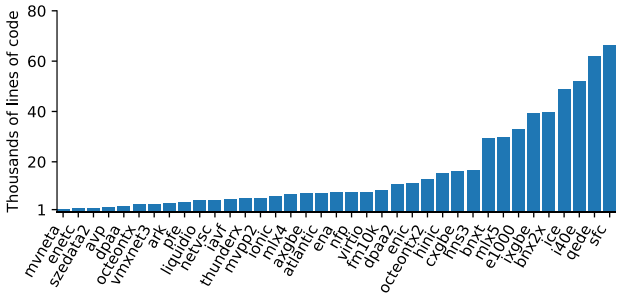

Figure 1. Number of lines of C code in the network drivers included with DPDK 20.02.

Software network stacks in modern operating systems are not adapted to network functions for three reasons.

First, traditional stacks use a push model: hardware uses interrupts to notify software of packet reception. If packets are infrequent, this is efficient. But in network functions, packets are frequent thus interrupt overheads dominate. The pull model, in which software continuously polls for packets, better fits network functions because it is efficient if most polls succeed, as is the case under high load.

Second, traditional stacks only access hardware through the operating system to provide isolation. Going through the operating system is an expensive operation, especially given the low time budget for each packet. But network functions typically run alone, paying the performance cost of isolation without the associated benefits. Systems such as netmap [28] have shown this cost can be amortized by processing packets in batches.

Third, traditional stacks allow to manage packet buffers with complete flexibility. This is convenient for general-purpose programs but hinders optimizations in the network stack. Network functions have restricted and well-defined behavior, yet they pay the performance cost of flexibility. Systems such as Windows Registered I/O [24] have shown that decreasing flexibility can increase performance.

Kernel-bypass stacks, which allow programs to access hardware directly instead of going through the operating system, arose from the need for different tradeoffs. These stacks also focus on polling instead of interrupts, on tighter control of packet buffers, and on processing packets in batches. The de facto standard kernel-bypass stack is DPDK, the Data Plane Development Kit [5].

Drivers, network or otherwise, have a poor reputation among software developers because of the challenges of hardware interactions and the lack of documentation.

Developers can only rely on specifications released by manufacturers to know how hardware behaves, and these specifications are not always public. Reverse-engineering hardware is infeasible without special equipment, unlike software. Since drivers are often exclusively maintained by hardware manufacturers, driver developers do not need to publicly document their code. Bug-finding efforts have shown that driver code is far from bug-free [21, 25].

These problems lead developers to think of drivers as mystical black boxes. But drivers are a fundamental part of the network stack; their correctness and performance are upper bounds on the entire stack.

As an example of driver complexity, the network drivers in DPDK, which supports many different types of hardware, all have at least 1,000 lines of code, as we show in Figure 1, with the largest one being over 66,000 lines of code.

Emmerich et al. [10] showed that network drivers can fit in under 1,000 lines of code, though their driver focuses on educational value and not performance or correctness.

3. Background on network cards

In this section, we summarize the architecture of modern Network Interface Controllers, or "NICs" for short, which is necessary to understand network driver design.

While NICs are diverse, the core concepts are similar. We estimate that, out of the 44 families of physical or virtual NICs supported by DPDK 20.02, this section applies to 40 of them. The remaining ones are three FPGA-based cards and one proprietary virtual NIC.

Communication between CPU and NIC uses three channels: PCI registers, NIC registers, and RAM.

PCI registers are stored on the NIC and accessed by the CPU using port-mapped I/O. The CPU only uses them for the first stage of NIC initialization.

NIC registers are stored on the NIC and accessed by the CPU using memory-mapped I/O. Their latency is an order of magnitude higher than RAM [19], making them a performance bottleneck.

RAM is the main shared storage. The CPU accesses it as usual, and the NIC uses Direct Memory Access, or "DMA" for short, to transfer data into it. RAM holds packet buffers and metadata. The CPU and the NIC are not notified when the other has modified RAM; if they want to be aware of changes, they must poll RAM or use a side channel.

The packet descriptor is the main NIC data structure, containing a pointer to a data buffer and some metadata. The metadata typically contains required fields such as the packet length, and optional fields such as whether to use advanced hardware offloading features.

Software chooses the total number of descriptors when initializing hardware. Descriptors are given from the CPU to the NIC to issue commands, such as packet transmission, and given by the NIC back to the CPU when the associated command has finished. Different NICs have different ways to manage descriptor ownership, such as flags in metadata.

Software can change the buffer pointer before giving a descriptor to the NIC. This lets developers implement buffer pools, to reuse descriptors without losing received data. This is useful for cases such as TCP, where packets must be kept until an entire message has arrived.

Reception and transmission are the core operations of network cards, and work in symmetric ways.

To receive packets, the CPU gives descriptors to the NIC indicating where to deposit packets in memory. The NIC gives descriptors back when it has received packets. The NIC sets descriptor metadata to indicate the packet length and other such information.

To transmit packets, the CPU gives descriptors to the NIC indicating where packets are in memory, and the NIC gives descriptors back once it has transmitted packets. The metadata is set by the CPU, to inform the NIC of the packet length and other such information.

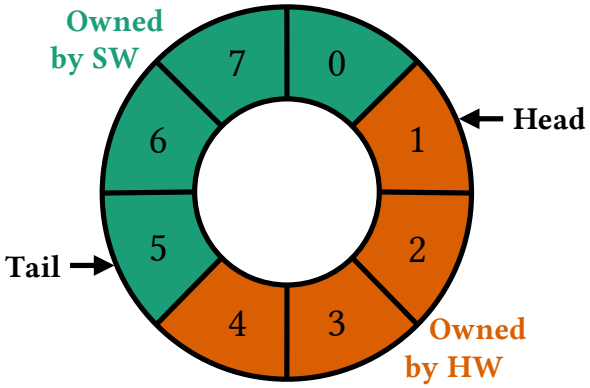

Figure 2. A descriptor ring with 8 elements; the head is 1 and the tail is 5, thus hardware owns elements 1, 2, 3 and 4.

Descriptor rings are the main mechanism for descriptor ownership in modern cards. We present here their inner workings in Intel's 82599 NIC as a concrete example.

A descriptor ring is composed of a region of memory, a head pointer, and a tail pointer. The memory is in RAM, while the pointers are NIC registers. Descriptors between the head, inclusive, and the tail, exclusive, belong to the NIC. Other descriptors belong to the CPU. If the head and tail are equal, the CPU owns all descriptors. We present an example ring in Figure 2.

Since descriptors start in an unknown state, descriptor metadata has a "Descriptor Done" flag to let the CPU know whether a descriptor has been processed by the NIC or was just never initialized.

The CPU gives descriptors to the NIC by clearing their "Descriptor Done" flag and incrementing the tail pointer. Since the tail pointer is a NIC register, the NIC immediately notices the change. The NIC gives descriptors back to the CPU by setting the "Descriptor Done" flag in metadata and incrementing the head pointer. To know when a descriptor has been given back, the CPU polls the metadata.

The head and tail can only be incremented, though they can be incremented by more than 1 to give descriptors in batches. Decrementing is forbidden since it would logically be an attempt to steal descriptors.

NIC queues are a hardware mechanism to allow for parallel packet processing. A queue consists of a descriptor ring and some configuration. The NIC places all received packets in the first reception queue by default; developers can configure the NIC to route packets to a queue based on packet headers, such that packets belonging to the same logical flow are routed to the same queue.

For transmission, all queues behave in the same way, without flow tracking: packets added to any transmission queue are sent to the wire regardless of which queue it is.

Queues allow multiple CPU cores to handle packets without having to synchronize NIC accesses, increasing software scalability.

4. Simplifying the driver model

In this section, we present the existing kernel-bypass driver model, and our proposed simplification of it for common network functions.

The driver model in modern frameworks such as DPDK is based around reusing a fixed set of packet buffers, in order to avoid the overheads of memory allocation. We formalize this model as a diagram in Figure 3, where the overall system is a set of first-in-first-out buffer queues. Each conceptual "step" of the system is performed by one of three actors: the NIC, the driver, or the network function. In the initial state, all buffers are in the "Free" state, which represents unused buffers in the buffer pool.

The driver typically takes the first steps, by "allocating" buffers from the pool and giving them to the NIC for reception. This "allocation" refers to taking buffers from the pool, not creating new ones. If there are buffers in the "receiving" state, the NIC can transition them to the "received" state once it gets data from the network. The network function typically runs a polling loop to move buffers into the "processing" step. From there, the network function can choose to transmit the buffer, possibly after modifying it. The NIC will then send out the buffer contents to the network and move the buffer to the "transmitted" state. The driver moves transmitted buffers back to the "free" state at well-defined points, for instance when there are too few free buffers left. The network function can also choose to keep the buffer for later, or to "drop" it and return it to the pool. The network function can also allocate buffers from the pool and process them like received buffers.

Unlike classical driver models found in mainstream operating systems and exposed to programmers in libraries such as BSD sockets [29], the system is closed: none of the actors can insert buffers into the system from the outside, such as by asking the operating system for memory. Actors cannot remove buffers either, though the network function is allowed to keep buffers indefinitely by using its "keep" transition to reorder buffers in the "processing" state.

The reason for a closed system is performance: buffer allocation and deallocation are expensive. This is not only due to general software issues such as the overheads of keeping a "free list" of memory blocks, or the cost of asking the operating system for more memory, but also to an issue specific to drivers: memory pinning. The driver gives physical memory addresses to the network card when specifying buffer addresses. If the operating system were to change which physical page backs a virtual page used by the driver, the network card would not see the change and write to the wrong page. Thus, the operating system has to be informed of which memory is used for buffers and give it special treatment. While modern hardware can use I/O memory management units to allow devices to address virtual memory, there is a cost to changing I/O memory mappings.

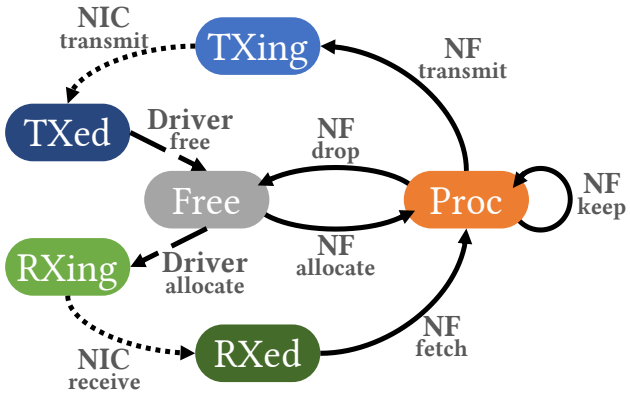

Figure 3. Diagram of the kernel-bypass driver model. Each box is a queue, each arrow is a step moving one packet from one queue to another. Steps are annotated with their actor and their name. "RX" is reception, "TX" is transmission, "Proc" is processing, and "NF" is network function.

This model provides flexibility to network functions: they can keep buffers aside to reassemble messages from high-level protocols such as TCP, and can allocate buffers from the pool in response to non-network events such as timers indicating a request needs to be retried.

The model also lends itself well to concurrency: the "free" queue is the central element shared by any number of reception, transmission or processing queues. A network function can receive and transmit packets from multiple NICs, and it can use multiple processing queues that each communicate with different reception and transmission queues on the same NIC to process packets concurrently and increase overall throughput.

But this flexibility comes at a cost: the steps that the network function can perform besides transmission introduce forks in the path of packet buffers. This requires buffer management within the "free" queue, including support for concurrent accesses. It also requires the driver to implement a policy for buffer freeing and allocation, adding complexity to the overall system.

The model additionally introduces a failure case that is not fundamental to the concept of a network function. If there is a state within the processing logic in which any buffer is kept, and the only way to get out of that state is to receive new data, the system will only make progress if there are buffers outside of the processing queue, which is not guaranteed. Reasoning about the existence of such a state requires reasoning about the invariants that hold in the network function code across packets.

This flexibility is not always needed: some of the network functions that power the backbone of the Internet, such as IP routers or Ethernet bridges, process packets one by one, never keep buffers aside, and never allocate buffers. Overall, they are conceptually simpler than the general case of a network function, yet they must currently pay the price of driver flexibility they do not use.

We propose a new driver model designed for common network functions that do not need the flexibility provided by existing models. It is based on two key insights: we can remove the buffer pool altogether, and we can implement buffer drops on modern NICs without the theoretical branch they introduce, minimizing the amount of state that the driver must keep track of.

Our model is designed to be as simple as possible, thus improving correctness and performance. Its simplicity makes it easier to formally or informally reason about and requires less code and simpler code to implement.

Our model is a subset of the existing model: as shown in Figure 4: the core differences are that it has no pool of free buffers, and does not allow network functions to keep buffers. The driver moves transmitted buffers directly to the reception queue, and the network function must choose to either transmit or drop received packets. This simplifies the driver by giving it only one choice when transmitting a packet: recycling transmitted buffers to the receiving queue now or later. Removing the buffer pool also makes progress easier to reason about: the software can only halt if the driver does not recycle buffers when the receiving queue is empty, or if the network function halts. While termination is impossible to prove in the general case due to the halting problem [31], network functions have strict performance requirements, thus their code is unlikely to have loops whose termination is not obvious because such loops could be performance bugs.

Our model minimizes state by combining reception, processing, and transmission into a single logical descriptor ring containing all buffers, without the need for any other data structure. While it is implemented using one reception ring and one transmission ring, the driver mirrors the head of the transmission ring to the tail of the reception ring, thus ensuring that buffers that have finished transmitting are reused for reception without any intermediate steps.

The key hardware feature that allows this is called "null transmit descriptors": as its name implies, it allows some descriptors in a transmission ring to have no effect. Packet drop is thus a special case of packet transmission, which removes the fork in buffers' paths and allows for a regular buffer flow. For instance, a network card can implement this by dropping packets whose length in metadata is zero.

The driver's job consists of three tasks: move buffers from the "received" queue to the "processing" queue when the network function asks for a packet, move buffers from the "processing" queue to the "transmitting" queue when the network function asks to transmit or drop its current packet, and recycle buffers from the "transmitted" queue to the "receiving" queue to ensure the "receiving" queue is never empty. Since this last operation is not a response to a specific input, the driver must choose when to perform it, for instance once every few transmitted packets.

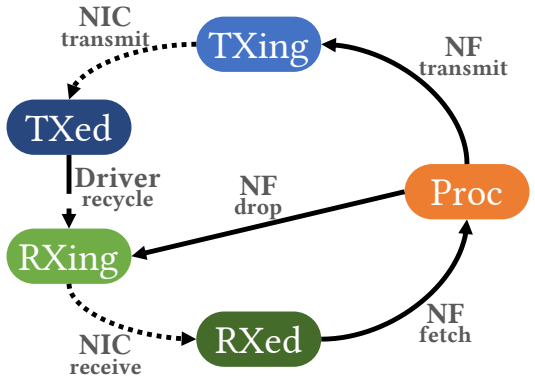

Figure 4. Diagram of our proposed driver model. Semantics are the same as in Figure 3.

Our model supports multiple outputs by using multiple transmission rings and making the driver synchronize their state. That is, the driver must set the tails of all transmission rings at the same time and use the earliest head in all rings as the head to mirror to the reception tail. Transmitting a packet when the driver has multiple outputs conceptually maps to transmitting it on some outputs and dropping it on all others; all rings still have a descriptor pointing to the buffer, but that descriptor is null in some of the rings. This may cause packet drops if an output link is too slow, in which case the entire ring will be used for transmission with no space left for reception. The same could happen in a traditional model if all buffers in the pool were used for transmission due to a slow output.

Multiple inputs can be handled concurrently: while the same processing queue cannot have multiple inputs, since it is not possible to synchronize the state of reception rings, the entire system can be duplicated so that there is one reception queue per input, one associated processing queue, and any number of synchronized transmission queues. Modern NICs have hundreds of queues, thus it is not a problem to use one transmission queue per input.

This does not mean our model requires parallelism: a single thread of execution can implement many instances, which are thus concurrent but not parallel.

Our model is amenable to parallelism: multiple threads of execution can run in parallel, each implementing any number of instances, without having to synchronize any state. Only the state of the rings within an instance needs to be kept in sync. This is similar to existing models.

The key limitation of our model is the flip side of its strength: since network functions must process buffers one by one without keep any aside, they cannot reconstruct multi-packet messages without copying buffers that arrive out of order. Thus, while core functions such as routing and network address translation can be implemented with our model, one cannot terminate TCP connections or otherwise reassemble fragments without copying buffers, which is an expensive operation given modern network speeds.

5. Implementing our new model

In this section, we describe an implementation of our driver model for the Intel 82599 NIC [12] which we call "TinyNF", short for "Tiny Network Function".

TinyNF's goals are to be easy to reason about and fast. The former is different from "correct" because it is hard to tell whether a driver operates as expected without hardware schematics, since the data sheet may be incorrect. However, we want to make it simple enough that it is not a bottleneck in network function verification efforts.

For simplicity, TinyNF processes buffers one at a time: there is always at most one buffer in the processing queue. One key hypothesis in this project was that TinyNF could be fast without explicitly processing packets in batches.

The keys to TinyNF's performance the avoidance of any operation that is not absolutely required, and the use of a few small but surprisingly effective scheduling algorithms for synchronizing queue state.

TinyNF avoids unneeded work, even metadata copy. Because each buffer always belongs to exactly one queue, and because queues are ordered, it is enough to set the buffer pointers at initialization time and never change them afterwards. Moving a buffer from one queue to another only requires writing to the source head and destination tail.

There are fewer delimiters in practice than in theory since some of them are implicit, as shown in Figure 5. The "transmitted" head and tail are the "receiving" tail and "transmitting" head, respectively. Similarly, the "received" head and tail are the "processing" tail and "receiving" head. While there is technically a "processed" queue that does not exist in the conceptual model, its head and tail are the "transmitting" tail and "processing" head respectively. The "processing" tail does not need explicit tracking, because it is always either one buffer ahead of the head or equal to it, due to the one-packet-at-a-time constraint.

TinyNF avoids reading from NIC registers entirely after initialization. To check for received buffers, the "descriptor done" metadata flag of the descriptor at the processing tail is enough. To check for transmitted buffers, the 82599 NIC provides a "transmit head write-back" feature: software can request hardware to write the transmit head to RAM after hardware has finished transmitting a buffer.

TinyNF cannot avoid updating the receive and transmit tails, which are NIC registers and thus slower than RAM, but it can avoid doing so after every packet. Updating the receive tail, which moves buffers to the "receiving" queue, is only necessary once every few transmitted buffers since reception continues working as long as there are buffers in the queue, even if there are less than there theoretically could be. Updating the transmit tail is necessary for buffers to be transmitted to the network, but this can be done once every few transmissions, or when there are no packets to receive and thus no other work to do.

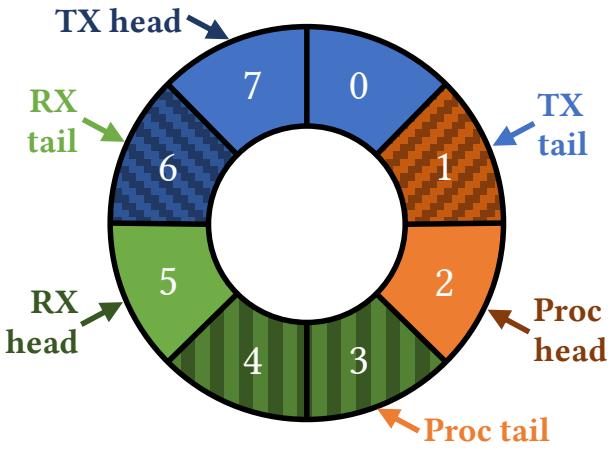

Figure 5. Logical ring composed of reception, processing and transmission queues. "In progress" queues are light, "done" ones are dark and shaded. Heads and tails refer to "in progress" queues, but implicitly delimit the others.

TinyNF carefully schedules operations to minimize the amount of communication between software and hardware. This improves overall latency and reduces the fraction of PCIe throughput used for metadata.

Two operations can be scheduled together: asking the NIC to update the transmission tail and checking for such updates to recycle buffers. The request is made with a bit in transmission metadata, and the check is made by reading the value that the NIC wrote to RAM via DMA. TinyNF schedules both operations once every 64 packets. The check will thus see the update that was requested 64 packets ago.

The most important scheduling decision is updating the transmission tail: frequent updates decrease latency by making the NIC aware of packets sooner, but they increase throughput by performing less book-keeping. Networking stacks such as DPDK solve this with adaptive batching: they check for multiple received buffers at a time up to a limit, let the network function process them all, then update the transmission tail. This theoretically allows drivers to make better scheduling decisions because they have more data: they know how many packets have arrived, rather than whether there is at least one packet.

TinyNF's one-packet-at-a-time model is incompatible with batching, thus we chose an algorithm based on past data instead. TinyNF updates the transmission tail either once every few transmitted packets, or as soon as there are no packets to receive since this likely indicates there is time to perform this expensive operation. This keeps the period short under low load, avoiding latency spikes, but allows for longer periods under high load, avoiding throughput drops, without looking at packets beyond the current one.

Overall, TinyNF is around 550 lines of C code, and its only dependency is a 300-line environment abstraction. It runs entirely in user mode, without kernel dependencies.

6. Evaluation: TinyNF for verification

In this section, we evaluate two hypotheses about TinyNF: its simplicity should (1) make it easier to reason about and (2) make it faster than other verified drivers.

We evaluate TinyNF by using it in the formally verified network functions of the Vigor [33] project. Vigor verifies the entire software stack, including the network function code and the network card driver.

Vigor does not need DPDK's flexibility: it is focused on network functions that form the Internet's backbone, such as Ethernet bridges and IP load balancers. This makes Vigor network functions good candidates to evaluate TinyNF.

Vigor uses DPDK for performance, but it cannot take full advantage of DPDK's optimizations either. Vigor's use of DPDK allows its network functions to outperform those written using traditional networking APIs that go through the kernel to receive and transmit packets. However, some DPDK optimizations such as batching and vectorization are currently out of the reach of automated formal verification. Thus, the driver formally verified by Vigor is the subset of the DPDK driver that can be automatically verified, not all of the driver.

TinyNF makes Vigor network functions 8x faster to verify, as we show in Table 1. We ran verification on two Intel Xeon E5-2690 CPUs at 2.90 GHz, totaling 32 cores.

Vigor verification has two steps: first Vigor symbolically executes the network function code to find all paths, then it validates each path using a theorem prover, which can be done in parallel. Both parts of Vigor verification are faster with TinyNF for the same main reason: symbolic execution does not need to explore DPDK's complex stack, thus it takes $1/5^{\text{th}}$ the time and yields $1/7^{\text{th}}$ the number of paths. Individual paths are also faster to validate since they have less code, though this is less pronounced since validation focuses on network function code, not driver code.

The most drastic change is in the load-balancer, due to its more complex paths that involve more data structures: its total verification time on our machine goes down from ~1h45min to ~14min. This allows full-stack verification to be used as part of development, such as verifying every code change, as opposed to being for special occasions.

	DPDK		TinyNF	
	Sym. ex.	Validation	Sym. ex.	Validation
NAT	337s	149 × 83s	63s	20 × 73s
Bridge	527s	312 × 89s	104s	39 × 77s
LB	731s	297 × 620s	161s	51 × 425s
Policer	392s	190 × 90s	75s	25 × 76s
FW	323s	140 × 83s	61s	20 × 68s

Table 1. Verification time statistics for the Vigor network functions using DPDK and TinyNF.

TinyNF is $1/11^{\text{th}}$ the code of the DPDK driver and has exponentially fewer paths, as we show in Table 2, which explains why the improvements in verification time are so drastic. We measured the code complexity of TinyNF and of the verified subset of DPDK's driver. We manually counted paths, so that we could define them in terms of the public parameters: the arguments passed in the code, and the choices made at DPDK build time when picking a data structure implementation. Automating this using symbolic execution would have only found the number of paths given a concrete configuration. When counting paths, we assume that NIC hardware behaves as per its data sheet.

To show the effect of a change in driver model and not only in implementation, we also included the "Ixy" driver by Emmerich et al. [10], a simplified implementation of DPDK's design for educational purposes that does not aim for comparable performance. As expected, TinyNF and Ixy use similar amounts of code to initialize, since they both use a limited set of NIC hardware features. However, TinyNF has less code and exponentially fewer paths than Ixy in the reception and transmission functions that form the core of the driver, providing more evidence in favor of our model.

We note that the number of paths can change based on programmer decisions: using Boolean expressions rather than conditionally executed code can lower the number of paths, such as writing *x = c ? y : x;* instead of *if (c) { x = y; }* in C. We could have used this to bring down the number of paths in TinyNF's transmission function to 4, without any exponent regardless of the number of output links, but chose not to as such code is compiled to conditional move instructions which have poor tail latency on our machines.

	Init.		Reception			Transmission		
	#funs	*#LoCs*	*#funs*	*#LoCs*	*#paths*	*#funs*	*#LoCs*	*#paths*
DPDK	115	3204	5	136	$1 + A_F + 288A_S$	5	122	$(8 + 14(F_F{}^T + P((F_S + F_F)^T - F_F{}^T))^O$
Ixy	14	279	1	63	$1 + A_F + A_S$	1	53	14^O
TinyNF	4	245	1	17	3	1	29	$2 + 2^O$

A_S, A_F and F_S, F_F: Number of success and failure paths in packet allocation and freeing respectively; Ixy's freeing cannot fail
P: Number of paths in the "put buffers back" operation of the DPDK memory pool in use
T: DPDK parameter for the transmit descriptors write-back threshold, must be >0 O: Number of output links

Table 2. Number of functions, lines of code and paths in DPDK, Ixy and TinyNF drivers for the Intel 82599.

TinyNF makes fewer assumptions on its environment than DPDK. Vigor makes assumptions about the behavior of two components: DPDK data structures and operating system functions.

One fundamental issue with DPDK's driver, even in the verified version, is its need for a data structure to hold free packet buffers. This leaves two options for verification: use a simpler but slower data structure that can be verified or assume that a faster but unverified data structure is correct. Unlike DPDK and TinyNF, there is no evidence that simpler data structures can match their more complex counterparts in performance. In fact, the opposite is true: data structure contracts are already simple yet popular implementations become more complex with time, such as a 2500-line change in Java 8 to make the hash map more resilient to collisions [15]. By comparison, Vigor's verified map has less than 300 lines of code in its entirety.

Another issue with DPDK's driver is in the amount of assumptions it makes about operating system functions. When verifying network functions running on Linux, Vigor replaces these functions during symbolic execution with custom models. This ensures DPDK calls operating system functions correctly according to Linux's documentation, such as by validating the order and arguments of function calls. The models then return symbolic values that cover the range of documented behaviors. But there is no formal specification for these functions, much less a formal proof that the Linux implementation is correct. Thus, Vigor needs to assume the correctness of dozens of models for its proof on Linux. This can be avoided by using a custom operating system, at the cost of losing Linux tools and features such as multitenancy and scheduling. TinyNF needs much less from its environment, drastically reducing the number of assumptions even on Linux.

TinyNF is easier to analyze than DPDK, since it only needs standard C. DPDK uses non-standard extensions to give hints to the CPU and compiler, such as prefetching memory and vectorizing loops. TinyNF does not need any such hints; the driver does not even use the standard library directly, going through a small environment abstraction layer instead.

This standards compliance makes TinyNF analyzable "out of the box" with most tools and allows future tools to support TinyNF without special treatment. This includes symbolic execution engines such as KLEE [4], which Vigor uses and extended to support DPDK code, and manual provers such as VeriFast [14], also used by Vigor. We think this will accelerate networking research in drivers and functions by making it easier to develop new techniques and tools. For instance, TinyNF's simplicity and small size makes it amenable to a proof of functional correctness given a hardware specification, which would improve upon Vigor's proof of memory safety through hardware models.

	DPDK			TinyNF		
	Tput	**Latency** *(µs)*		**Tput**	**Latency** *(µs)*	
	(Gb/s)	*50%*	*99%*	*(Gb/s)*	*50%*	*99%*
NAT	1.99	4.04	4.77	3.69	3.92	4.25
Bridge	2.65	3.97	4.50	5.82	3.93	4.23
LB	2.22	4.01	4.63	6.66	3.90	4.24
Policer	2.96	3.88	4.32	9.53	3.83	4.24
FW	2.65	3.97	4.49	8.14	3.88	4.24

Table 3. Single-link throughput and latency with 1 Gb/s background load of Vigor functions on DPDK and TinyNF.

TinyNF improves the throughput of Vigor network functions by 160%, with 2% less median latency, as we show in Table 3. 99th percentile latency decreases by 7%.

To measure performance, we used two machines in a setup based on RFC 2544 [26], with a "device under test" running a network function and a "tester" running the MoonGen packet generator [9], which can measure latency using NIC timestamps. Both machines run Ubuntu 18.04 on two Intel Xeon E5-2667 v2 CPUs at 3.60GHz with power-saving features disabled and have two Intel 82599ES NICs, using only one port per card to ensure PCIe bandwidth is not a bottleneck. We measure throughput using minimally sized packets. Our workload fills the internal flow table of the network functions to 90% of their capacity. Measuring latency with MoonGen instead of on the device under test allows us to capture the latency of NIC register writes as well as the effects of drivers' NIC configuration. This setup is similar to the one used to originally evaluate Vigor, and can replicate Intel's DPDK performance numbers [7].

We replicate Vigor's benchmark setting: measuring the max throughput that a Vigor network function can achieve with less than 0.1% loss, in a single direction, as well as the latency with 1 Gb/s of background load.

Vigor's NAT gets the lowest throughput improvement; this is because its bottleneck is not the driver but computing packet checksums since it has to modify packet headers. To confirm this, we tried modifying the DPDK version of the NAT to use batching: this results in the same throughput as the TinyNF version of the NAT, confirming that the driver is unlikely to be the bottleneck.

In summary, both of our hypotheses are validated: TinyNF is easier to reason about in terms of code quantity and code complexity, and network functions using TinyNF are faster than the same functions using DPDK's verified subset. Thus, TinyNF allows developers to formally verify their network functions in less time, get more correctness guarantees, more than double the functions' throughput, and lower the functions' median and tail latency.

7. Evaluation: TinyNF in general

In this section, we compare the performance of TinyNF and DPDK for general purpose network functions, regardless of verifiability.

We use the same benchmark setup as in the previous section, but this time use both directions for throughput, for a maximum of 20 Gb/s. We keep throughput symmetric during the benchmarks, i.e., if a function cannot handle a given load, we reduce the load of both directions by the same amount and retry. We then measure the latency at load increments of 1 Gb/s to paint a clear picture of the function's overall performance profile.

TinyNF can outperform a fully optimized DPDK setup, as we show in Figures 6 and 7 using a traffic policer as an example. We compare the Vigor policer using TinyNF as its driver to the same code using either "unbatched" DPDK, which is the simpler version used by Vigor, or "batched" DPDK, which is the standard way to use DPDK that enables optimizations such as adaptive batching and vectorization. We also implemented a 2-core parallelization of the policer for all three variants. We chose the policer because, by design, traffic in one direction is independent of traffic in the other, which means it admits a trivial 2-core parallelization for our experiments. We are not proposing a new way to parallelize network functions, but merely showing that TinyNF can be parallelized in a similar way to existing drivers. This also shows how much improvement parallelization can bring compared to batching.

Using TinyNF, the policer achieves better throughput than using batched DPDK, with an even starker difference when using two cores. The bottleneck that prevents the dual-core TinyNF version of the policer from reaching line rate is the frequent reads from the CPU time, which it needs for flow expiration.

TinyNF leads to better latency at low and high loads but worse latency in the middle, especially the 99th percentile latency. Looking at individual data points, which we show in Figure 8, the TinyNF-based policer has lower latency in some cases, but this advantage is lost in the tail latency. We believe this is a case where DPDK's batching shines: it can detect "gaps" between packets, in which updates to the transmission tail do not compete with packet processing, by looking at how many packets there are in the queue.

Finally, since we had to modify the policer code to use TinyNF, we wanted to see whether the same performance benefits could be obtained without code changes. We wrote a compatibility layer that implements some of the DPDK API on top of TinyNF. The layer cannot implement all of the DPDK API, by design, but can replace DPDK for functions that fit the TinyNF model by changing an environment variable at compile time. The compatibility layer allows for 1% more maximum throughput than batched DPDK, at the cost of increased latency.

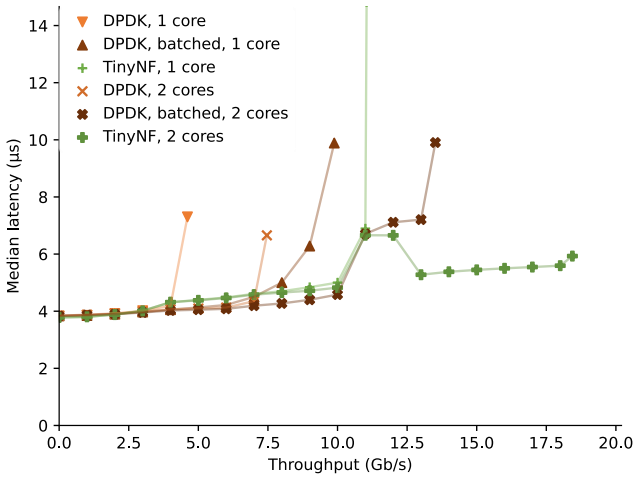

Figure 6. Throughput and median latency of a traffic policer using DPDK with and without batching, TinyNF, and 2-core versions of all three.

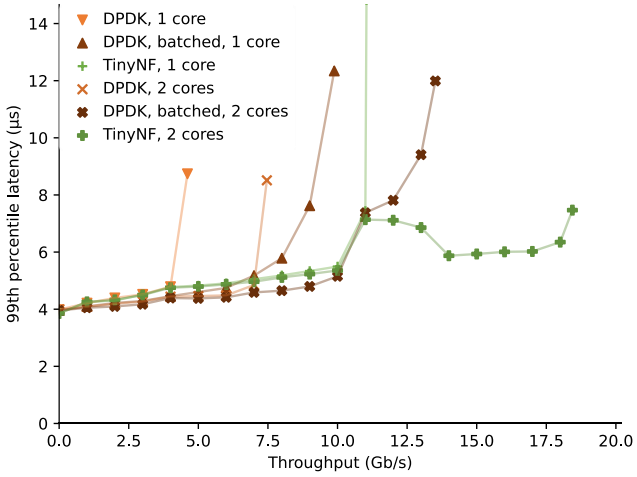

Figure 7. 99th percentile latency version of Figure 6.

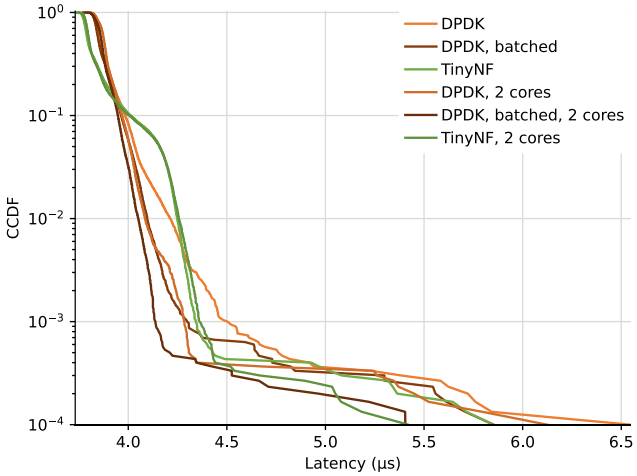

Figure 8. Complementary cumulative latency distributions of a traffic policer using the same alternatives as Figure 6, with 1 Gb/s background load.

A no-op function can handle more throughput with DPDK than with TinyNF, even though the opposite holds with real functions. We reached this surprising conclusion by benchmarking DPDK's "testpmd" built-in application, which DPDK developers use in performance reports [7] to benchmark driver speed. We configured testpmd to update packets' MAC address to provide some realism. Using our setup, both TinyNF and DPDK in its batched mode could saturate two 10 Gb/s links, as we show in Figure 9. We also included the Ixy driver [10], which performed admirably given its educational purpose but could not sustain line rate even with batching.

Since our setup was bottlenecked by link capacity, we chose to lower the CPU frequency to 2 GHz and re-run the benchmark. In this setup, DPDK can reach 97.5% of line rate while TinyNF peaks at around 92.5% of line rate, as we show in Figure 10, though its latency is lower.

We believe the bump around 11 Gb/s is due to hardware issues, since it appears in three independently written drivers and in both a no-op and a nontrivial function.

This result is interesting, since the no-op benchmark is the one used by DPDK developers to measure their progress when optimizing DPDK's performance. If this benchmark does not accurately represent driver performance on real network functions, the DPDK developers may believe they are improving DPDK's performance but do the opposite.

To explain this finding, we started by plotting the no-op function's latency in more detail. We did this because of an observation we made while running the other benchmarks: TinyNF's performance appeared more stable than DPDK's, yielding more consistent results across runs, such as never dropping packets under high loads whereas DPDK would sometimes drop a few packets per million.

As expected, TinyNF has a more stable latency profile than DPDK: without background load, TinyNF's latency remains low up until the 99.9th percentile, whereas DPDK's latency starts jittering before this, as show in Figure 11. We stop at the 99.99th percentile because Primorac et al. showed that NIC timestamping is not accurate after that point [23].

This measurement highlights a key issue with DPDK's driver model: the driver has to manage buffers explicitly instead of merely moving them from one queue to the next, which leads to a distinct bump in latency before the 99th percentile. The same holds for Ixy, since it uses the same driver model as DPDK.

We used the toplev microarchitectural measurement tool [22] to investigate bottlenecks in DPDK's driver when running the Vigor policer. While the tool indicates that the policer is bottlenecked on memory writes, there is no single write that dominates. Some of the memory writes that take the most time are fundamental to DPDK's design, such as moving buffer pointers to and from the buffer pool, while others could be removed at the cost of some functionality, such as writes to packet buffer metadata.

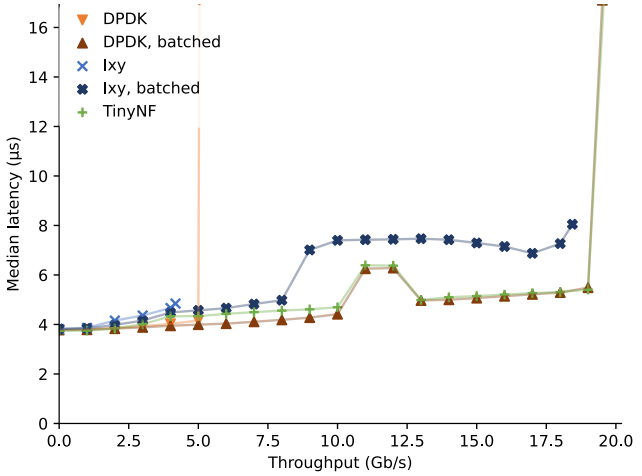

Figure 9. Throughput and median latency of DPDK's no-op function with and without batching, a port of it on TinyNF, and a port of it on Ixy with and without batching.

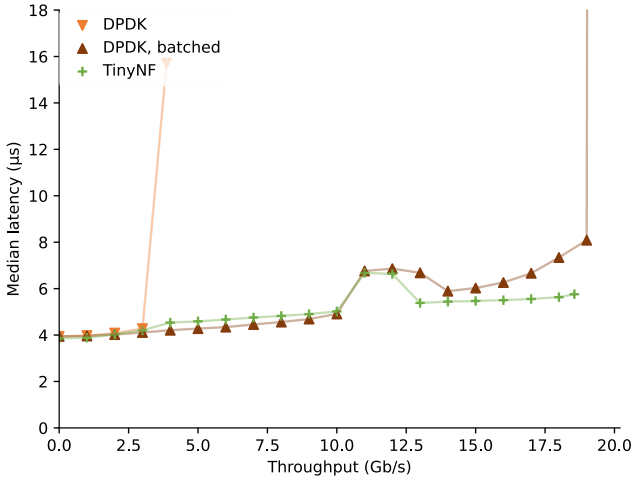

Figure 10. Same benchmark as Figure 9 but with the CPU capped to 2 GHz. We do not show Ixy since it could not sustain line rate even at full CPU speed.

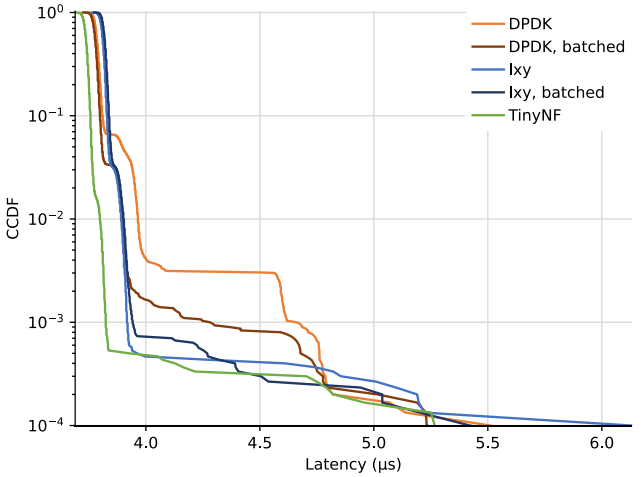

Figure 11. Complementary cumulative latency distributions of the no-ops from Figure 9 without background load.

TinyNF slows down less when running real functions because its instruction-level parallelism has room to grow and it interferes less with the CPU's caches. We reached this conclusion after measuring low-level CPU counters using libPAPI [30], in particular the number of cycles, instructions, and cache hits per packet at 20 Gb/s.

Before going further, we must caution against over-interpreting our results, in particular absolute numbers of cycles. To measure low-level CPU metrics, we instrumented network functions with code that copies counter values for later processing. This has overhead: reading performance counters uses cycles, and copying their values touches the CPU caches. Furthermore, due to the out-of-order nature of modern CPUs, accurately measuring cycle counts requires inserting serializing instructions to ensure past instructions have completed. Thus, measuring the cycle count increases it as it prevents the CPU from reordering some instructions. The measurement overhead is stable, so we measure it and subtract it from the measurements, but we cannot fully account for cache changes due to storing counter values, or for the effects of serialization. Because of this, cycle counts can only be compared to other functions on the same driver. Instruction counts and cache use can be compared globally.

We collected data by running network functions ten times collecting ten million packets each time. We intended to collect data in a single run, but noticed that some runs have a lower cache miss rate than others, despite using the same executable run in the same way on a CPU not otherwise used by the operating system.

We used four functions: a no-op function that does not even touch packets, one that writes a constant to the destination MAC address, one that sets the destination MAC address using a lookup table based on the source MAC address, and the Vigor policer. In our setup, the write function is faster on DPDK but the lookup one is faster on TinyNF. We report the measured cycles, instructions and cache hits in Table 4. We do not report main memory hits as they are negligible, around one in a million packets.

Two results stand out: the increase in instructions per cycle for TinyNF when running more realistic functions, and TinyNF's low cache use compared to DPDK.

TinyNF has low instruction-level paralelism in a no-op because the CPU is waiting for operations on descriptors and NIC registers, which cannot be executed out of order. On a more realistic function, the CPU executes the function instructions out of order, increasing efficiency, thus the slowdown is not linear. This is consistent with TinyNF's low latency in the reduced frequency benchmark: the frequency makes little difference when waiting for the NIC.

Batched DPDK, on the other hand, can execute multiple instructions per cycle even in no-ops, due to instructions for metadata and buffer management. Its use of vector instructions also helps keep a high instruction count per cycle by waiting for multiple descriptors in parallel without reordering. The slowdown when executing a real function is thus linear in the number of instructions, unlike TinyNF.

TinyNF also has a lower memory footprint than DPDK, thus realistic functions have fewer cache misses, an effect that cannot be observed in no-ops.

	IPC	**Cycles**		**Instrs**		**L1d hits**		**L2 hits**		**L3 hits**	
	50%	*50%*	*99%*	*50%*	*99%*	*50%*	*99%*	*50%*	*99%*	*50%*	*99%*
DPDK unbatched											
No-op	0.39	664	2140	258	3780	101	1300	8.94	103	1.00	82.0
MAC write	0.37	725	2220	267	3790	107	1300	10.3	102	2.00	85.0
MAC lookup	0.39	746	2180	287	3810	116	1310	10.4	96.1	3.00	96.0
Policer	0.66	866	2540	669	4130	331	1500	4.94	94.4	3.00	95.0
DPDK batched											
No-op	1.70	58.1	64.3	99.0	99.1	32.3	33.0	4.81	5.83	1.41	2.50
MAC write	1.68	63.9	70.1	107	107	36.3	37.0	4.74	5.65	2.66	3.62
MAC lookup	1.53	84.4	93.1	129	129	46.6	47.3	5.01	6.07	5.12	5.94
Policer	1.65	298	333	511	512	265	269	4.33	5.52	4.47	5.53
TinyNF											
No-op	0.12	289	683	35.0	53.0	7.87	16.7	4.51	11.0	0.00	1.00
MAC write	0.13	339	717	45.0	63.0	13.8	22.2	5.11	12.8	1.00	3.00
MAC lookup	0.18	360	734	65.0	83.0	19.7	29.7	8.99	14.9	2.00	4.00
Policer	0.49	490	883	297	308	125	144	11.0	23.0	2.00	4.00

Table 4. Low-level metrics. IPC is Instructions Per Cycle. Cycles and IPC are only comparable within the same driver, as explained in the main text. DPDK batched uses batches of size 32. Main memory hits are negligible and not shown.

8. Applicability

In this section, we evaluate the applicability of our model to real-world network function deployment. Did we strike a good tradeoff choosing not to support some functions to simplify the model? And is our model useful in the context of network function virtualization?

As previously explained, the core limitation of our driver model is that network functions cannot keep buffers aside for later use. For instance, they cannot reconstruct messages in TCP or other higher-level protocols. Our model targets network functions that do not need to do so because they logically handle packets one at a time.

Our model supports many well-known functions, though there is no standard list of network functions. Despite their increased importance in modern networking, there is no consensus on what is a "network function" and what is not. There have been attempts such as RFC 3234 [27] to classify "middleboxes", which are functions that are not crucial to the network, but to the best of our knowledge there is no commonly accepted list of network functions. We chose to use the list of functions from the ClickOS [18] paper, which were also used by the authors of Vigor [33] to estimate the applicability of their verification technique. We complement this list with our own knowledge, for lack of a more standard source.

Our driver model supports 13 of the 14 types of network functions listed in ClickOS: load balancing, DPI, NAT, firewalls, tunnel, multicast, BRAS, monitoring, DDoS prevention, IP proxies, congestion control, IDS, and IPS. The only one that our model cannot support without compromises is a traffic shaper, because shaping requires keeping packets to send them later in the desired traffic shape. Among the network functions not mentioned by ClickOS, our model can be used for Ethernet bridges, ARP clients and servers, DNS proxies, statistics collectors, traffic policers, and Google's Maglev [8] load-balancer.

However, our driver model cannot efficiently support functions based on entire TCP messages, since this requires keeping IP packets around to reorder and merge them into logical messages. Such functions include proxies and HTTP servers. While one could implement reordering by copying buffers before giving descriptors back to the hardware, this would hinder performance.

We believe our model is a good fit for network functions that form the backbone of networks, such as routing, load-balancing, NAT and DNS, access control and statistics. However, it is not suited to high-level functions that deal with entire connections or protocols that fragment packets.

Some requirements are orthogonal to our model. For instance, offloading checksums to hardware would remove the main bottleneck in the NAT we benchmarked. Any such feature that can be used by providing metadata to the NIC can be implemented in a driver using our model.

TinyNF can be used for virtualization, which is a key tool for the practical deployment of network functions [35]. Virtualization allows operators to deploy multiple network functions on the same physical machine, instead of having to dedicate an entire machine to a single function. They also provide an easier way to manage network functions, in the same way virtual machines ease software management.

We experimented with virtualization using Single-Root I/O Virtualization, or "SR-IOV" for short, a PCIe standard with which network cards can expose virtual network cards with the same packet-processing features as the physical card. The virtual machine monitor can let virtual machines access virtual devices directly, without surrendering control over the physical card. The physical card includes hardware to route packets to virtual cards based on packet headers, for instance by Ethernet address. The physical card can limit the rate at which each virtual card transmits packets and can prevent virtual cards from transmitting packets with a different source address than their own. Virtual machines thus gain the benefits of direct access without the ability to monopolize the link or lie about their network identity.

The Intel 82599's virtual cards do not support some of the physical features. Notably, using transmit head write-back causes virtual cards to hang, a problem not mentioned in the card's data sheet but already reported by the authors of Arrakis [20]. Another missing feature is legacy packet descriptors, which are simpler to use, though the data sheet calls this out. We wrote a version of TinyNF that does not use these features, making it slightly slower. The Arrakis authors estimated that the lack of transmit head write-back causes a 5% performance penalty

We used the same physical setup as before, but with 16 virtual functions on each of the two network cards, for a total of 32 virtual cards. Each virtual card has an Ethernet address, and physical cards route packets to virtual cards based on these addresses. The only code changes are due to the missing features mentioned above, as well as a few dozen lines of configuration. The functions forward each packet using a virtual card on the physical card opposite the one whose virtual card received the packet.

The Vigor policer handles 12.2 Gb/s of minimally-sized packets without loss when using TinyNF in this setup. A no-op function reaches 14 Gb/s. Both are bottlenecked by reading packet descriptors for packet fetches, as the data from packets and descriptors no longer fits in the L2 cache.

This experiment is only intended to show that our driver model is applicable to virtualized environments. With this number of devices, other concerns arise such as load skew across devices and non-uniform memory accesses, which we do not capture here. We believe TinyNF is as sensitive to these concerns as other stacks. In particular, the order in which the function checks virtual cards for packets matters. For instance, if packets mostly arrive on one card, checking the other cards for packets will limit performance.

9. Discussion

In this section, we present our main takeaways from this project, in the form of actionable recommendations for both researchers and practitioners.

Drivers are not a special category of software, and the line currently drawn between drivers and other kinds of software is neither well-defined nor helpful. Drivers should be considered just another kind of software system, one that is more focused on hardware than usual. The same techniques used in systems that handle requests can and should be scaled down to "drivers", instead of creating new vocabulary for one kind of software.

The common meaning of "driver" is a piece of code that has exclusive access to hardware and exposes a software API to programs who want to use it. However, software that does this is not always called a driver. Operating systems allows programs to access CPUs, including isolation and high-level APIs to access features such as clocks, but they are not commonly referred to as "CPU drivers", with the notable exception of Barrelfish [1]. The same can be said of higher-level frameworks such as Java or .NET, which offer an abstraction over low-level CPU details yet are not called drivers. This applies to other kinds of devices as well: code that lets programs run GPU shaders is called a driver, but code that lets programs to draw windows and buttons on the screen is not, even though it is also a way for programs to draw. The internal architecture of some systems does rely on "drivers" as an indirection to access hardware, but this is not relevant from users' point of view.

An example of overly specific vocabulary is "batching" in network drivers: a feature that improves performance by amortizing costs. It is really composed of three independent features: (1) getting multiple packets at a time from the NIC, gaining information about network load, (2) processing multiple packets at a time, allowing for vectorized code, and (3) giving multiple packets at a time to the NIC, amortizing the cost of NIC register writes. TinyNF shows that only (3) is required for high throughput, though (1) may be required to get consistently low latency. In fact, any developer that uses batching but does not explicitly keep track of network load or use vector operations is already implicitly aware of this. Amortizing NIC writes is similar to existing techniques such as buffering reads and coalescing writes in disk I/O.

The idea that drivers are a special kind of software is hindering research. Most systems for fast networking, such as ClickOS [18], DPDK [5], netmap [28], SoftNIC [11], and IX [2], reuse existing drivers, which are bottlenecks on their performance. Arrakis [20] uses custom drivers but focuses on interrupt-driven I/O, which strikes a different tradeoff. Ixy [10], is the only research driver we know of besides ours. It is odd to have more research operating systems than drivers: the former are by definition more complex as they contain at least one driver.

Isolation is required for low-level performance, just as modularity is required for high-level correctness. The best-effort approach of shared caches is no longer enough when interferences that cause even a low number of cache misses cause a noticeable performance difference, as is the case with fast networking.

One way to provide performance modularity is to run each part of a system on physically separate hardware, as in TAS [16]. This eliminates interference in per-core caches, at the cost of increasing resource use. It also increases the cost of communication between modules, in the same way protection rings eliminate functional interference between user and kernel mode at the cost of an expensive boundary between the two modes.

However, the current way to measure low-level metrics through special CPU registers cannot be isolated from the code under measurement. This is not an issue for most code, because the overhead of measurement is low, but it becomes an issue with nanosecond-scale code such as TinyNF.

One way to avoid measurement overhead is to use static instead of dynamic analysis, but this requires a hardware model. TiML [32] includes performance reasoning in a type system, and Bolt [13] infers performance metrics from the source code of network functions written in C. However, predicting cycle counts requires accurate hardware models. For instance, Bolt predicts instruction counts within a few percent of ground truth but is 300% off the true cycle count for typical workloads. Since hardware optimizations are considered a competitive advantage, perfectly accurate hardware models are unlikely to be made publicly available.

Standard benchmarks would improve the state of network function research. Other areas of research use benchmarks such as SPEC [3] to measure improvements on a widely-accepted scale. There is no equivalent for network functions, not even non-standard ones.

We chose to explore a new point in the design space of networking code based on our experience with networking research, but the main threat to this paper's validity is that we have no way to validate the usefulness of this design. It may be that real-world traffic looks more like the one used to benchmark Arrakis [20], for instance, in which Peter et al. came to the conclusion that handling operations in user mode entirely eliminates the need for even transmission tail update coalescing.

Unlike other domains in which one can substitute benchmarks with well-known publicly available targets, such as compiling the compiler itself to show optimization improvements, network functions are generally not public.

This problem is getting worse as hardware gets faster. With 100 Gb/s Ethernet becoming more popular, should we focus on handling minimally sized packets, with a budget of 6ns per packet, or should we assume that traffic is made up of packets in the hundreds of bytes, as Pigasus [34] does? We do not know.

Any benchmark, even if unrealistic, would improve the situation, which is that both industry and academia use no-op functions as a de facto standard. DPDK's performance reports [6] from Intel, Mellanox, and Broadcom all exclusively use no-ops, and research such as netmap [28] or SoftNIC [11] mostly use no-ops. But no-ops are not representative of either general or specific cases. Even overly specific benchmark suite would at least result in systems optimized for a real use case, instead of systems optimized for no-ops which are not useful to anyone.

Since the performance of infrastructure code interferes with the performance of application code in non-obvious ways, extrapolations from no-ops are not representative of actual performance. We believe that using any real network function as a standard benchmark would provide a data point from which one can extrapolate more credibly to other real functions. The chosen function would be closer to any other function than a no-op is in terms of how the infrastructure code influences its performance, regardless of how close it is in terms of functionality.

We started this project with the goal to close the gap between unverified and verified performance using the Vigor network functions as benchmarks. Had we measured no-op performance for TinyNF first, under the belief that it was representative, we would have come to the conclusion that it was worse than DPDK. This could have led us to make TinyNF more complex to "fix" its no-op performance, accidentally lowering performance for real functions in the process.

More formal hardware data sheets could speed up software development and reduce bugs, without the need to change the hardware. TinyNF's complexity mainly comes from the number of assumptions it makes about hardware. These are due to missing or incorrect data, which is a natural consequence of free-form data sheets.

Most of the data sheet errors could be avoided using the same kind of analysis performed by compilers today. For instance, the Intel 82599 NIC's data sheet [12] has typos in register names and even in the size of some register fields; these could be caught by consistency checks ensuring all referenced names are declared and all registers contain the right number of bits. Some registers are only documented within the list of registers and not in the explanations of the operations they are used for, requiring developers to read the entire data sheet to learn about them; these could be caught by a check for unused declarations.

It would be unreasonable to expect hardware engineers to always provide perfect data sheets or design bug-free hardware, in the same way that it would be unreasonable to expect software engineers to always write bug-free code. However, our experience is that most current bugs are low-hanging fruit that could be caught without inventing new analysis techniques, if data sheets written in a machine-readable format first.

Using the basic features of a modern NIC does not have to be complicated, despite the belief that hardware has become inherently harder to deal with than in the past. We examined the oldest driver we could find for a NIC of the Intel 8259x family, which is the so-called "apricot" driver [17] for the Intel 82596, released with Linux 1.1 in 1994. It contains 450 lines of code not including debug code, which is close to TinyNF's 550.

Most lines of code in TinyNF come from unused features that must be initialized anyway. For instance, software must clear packet filters and virtualization-related registers after resetting the hardware, unlike some other features that are left in a clean state by the hardware reset. This kind of issues is not a fundamental source of complexity but a hardware implementation detail. If the hardware could be fully reset in a single operation, TinyNF would have fewer lines of code than the old "apricot" driver. This overhead is not as visible in a driver such as DPDK's, whose complexity comes from the amount of features it supports.

We hope this paper serves as evidence that developing code that interacts with network cards is both interesting and rewarding, and that it is not as complex or difficult as is often believed. On the contrary, we found that developing our own driver made the development and verification of network functions easier, by removing all dependencies on complex external stacks and kernel-mode drivers.

Acknowledgements

We thank our shepherd Simon Peter for his useful feedback and guidance; the anonymous reviewers for their useful and detailed reviews; the anonymous artifact evaluators for their useful feedback on the code and experiments; Arseniy Zaostrovnykh, Akvilė Valentukonytė, Blagovesta Kostova, Katerina Argyraki, Lei Yan, Rishabh Iyer, Samuel Chassot, and Yassmine Abdrabo, for providing feedback on the ideas, paper, and code.

Availability

Our code is available at *github.com/dslab-epfl/tinynf*, and described further in the Artifact Appendix below. The code obtained the "Artifact Available", "Artifact Functional" and "Results Reproduced" badge from artifact evaluation and can thus be reused by others with confidence.

In particular, the TinyNF code can be used as a simpler and faster base for any network function that fits its model, or as a baseline to evaluate low-level networking code. The benchmarking scripts are independent of TinyNF and can be reused to measure the performance of network functions that use any framework or driver.

References

[1] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), 29–44.

[2] Belay, A., Prekas, G., Primorac, M., Klimovic, A., Grossman, S., Kozyrakis, C. and Bugnion, E. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4 (Dec. 2016). DOI:https://doi.org/10.1145/2997641.

[3] Bucek, J., Lange, K.-D. and v. Kistowski, J. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2018), 41–42.

[4] Cadar, C., Dunbar, D. and Engler, D. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), 209–224.

[5] Data Plane Development Kit: *https://www.dpdk.org/*. Accessed: 2020-01-21.

[6] DPDK - Performance reports: *http://core.dpdk.org/perf-reports/*. Accessed: 2020-05-26.

[7] DPDK Intel NIC Performance Report Release 20.02: *https://fast.dpdk.org/doc/perf/DPDK_20_02_Intel_NIC_performance_report.pdf*. Accessed: 2020-05-27.

[8] Eisenbud, D.E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W. and Hosein, J.D. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), 523–535.

[9] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F. and Carle, G. 2015. MoonGen: A Scriptable High-Speed Packet Generator. *Proceedings of the 2015 Internet Measurement Conference* (New York, NY, USA, 2015), 275–287.

[10] Emmerich, P., Pudelko, M., Bauer, S., Huber, S., Zwickl, T. and Carle, G. 2019. User Space Network Drivers. *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (Los Alamitos, CA, USA, Sep. 2019), 1–12.

[11] Han, S., Jang, K., Panda, A., Palkar, S., Han, D. and Ratnasamy, S. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report #UCB/EECS-2015-155. EECS Department, University of California, Berkeley.

[12] Intel 82599 10 Gigabit Ethernet Controller Technical Library: *https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/82599-10-gigabit-ethernet-controller/technical-library.html*. Accessed: 2020-01-21.

[13] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K. and Candea, G. 2019. Performance Contracts for Software Network Functions. *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), 517–530.

[14] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W. and Piessens, F. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *Proceedings of the Third International Conference on NASA Formal Methods* (Berlin, Heidelberg, 2011), 41–55.

[15] JDK-8023463: Improvements to HashMap / LinkedHashMap use of bins/buckets and trees: *https://bugs.openjdk.java.net/browse/JDK-8023463*. Accessed: 2020-09-08.

[16] Kaufmann, A., Stamler, T., Peter, S., Sharma, N.Kr., Krishnamurthy, A. and Anderson, T. 2019. TAS: TCP Acceleration as an OS Service. *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019).

[17] Linux 1.1.23. The "apricot" driver is in drivers/net/apricot.c.: *https://mirrors.edge.kernel.org/pub/linux/kernel/v1.1/*. Accessed: 2020-01-21.

[18] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. and Huici, F. 2014. ClickOS and the Art of Network Function Virtualization. *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (USA, 2014), 459–473.

[19] Neugebauer, R., Antichi, G., Zazo, J.F., Audzevich, Y., López-Buedo, S. and Moore, A.W. 2018. Understanding PCIe Performance for End Host Networking. *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), 327–341.

[20] Peter, S., Li, J., Zhang, I., Ports, D.R.K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4 (Nov. 2015). DOI:https://doi.org/10.1145/2812806.

[21] Pirelli, S., Zaostrovnykh, A. and Candea, G. 2018. A Formally Verified NAT Stack. *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks, KBNets@SIGCOMM 2018, Budapest, Hungary, August 20, 2018* (2018), 8–14.

[22] pmu-tools GitHub repository: *https://github.com/andikleen/pmu-tools*. Accessed: 2020-09-11.

[23] Primorac, M., Bugnion, E. and Argyraki, K. 2017. How to Measure the Killer Microsecond. *Proceedings of the Workshop on Kernel-Bypass Networks* (New York, NY, USA, 2017), 37–42.

[24] Registered Input/Output (RIO) API Extensions: *https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh997032(v=ws.11)*. Accessed: 2020-01-21.

[25] Renzelmann, M.J., Kadav, A. and Swift, M.M. 2012. SymDrive: Testing Drivers without Devices. *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), 279–292.

[26] RFC 2544 - Benchmarking Methodology for Network Interconnect Devices: 1999. *https://www.ietf.org/rfc/rfc2544.txt*. Accessed: 2020-05-26.

[27] RFC 3234 - Middleboxes: Taxonomy and Issues: *https://tools.ietf.org/html/rfc3234*. Accessed: 2020-01-21.

[28] Rizzo, L. 2012. Netmap: A Novel Framework for Fast Packet I/O. *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (USA, 2012), 9.

[29] sys/socket.h - main sockets header: *https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_socket.h.html*.

[30] Terpstra, D., Jagode, H., You, H. and Dongarra, J. 2010. Collecting Performance Data with PAPI-C. *Tools for High Performance Computing 2009* (Berlin, Heidelberg, 2010), 157–173.

[31] Turing, A.M. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. s2-42, 1 (1937), 230–265. DOI:https://doi.org/10.1112/plms/s2-42.1.230.

[32] Wang, P., Wang, D. and Chlipala, A. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017). DOI:https://doi.org/10.1145/3133903.

[33] Zaostrovnykh, A., Pirelli, S., Iyer, R., Rizzo, M., Pedrosa, L., Argyraki, K. and Candea, G. 2019. Verifying Software Network Functions with No Verification Expertise. *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), 275–290.

[34] Zhao, Z., Sadok, H., Atre, N., Hoe, J., Sekar, V. and Sherry, J. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2020).

[35] 2012. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action. Issue 1. ETSI.

Artifact Appendix

Abstract

The artifact of this paper contains the code of the "TinyNF" prototype, as well as scripts to run the various experiments used in this paper. The benchmarking scripts can be used for any network function, not only TinyNF.

Checklist

Program: driver and network functions
Metrics: throughput, latency, code complexity
Output: compiled network function including the driver
Experiments: as described in Sections 6, 7, and 8
Required disk space: 80 GB for low-level metrics, a few MBs for everything else
Expected run time: around half a day to run all available experiments, almost all of which is spent waiting
Public link: *github.com/dslab-epfl/tinynf*
Code license: MIT

Description

How to access: Use the link above.
Hardware dependencies: Two machines with Intel 82599 NICs, as mentioned in *experiments/ReadMe.md*.
Software dependencies: TinyNF currently supports Linux only. A few standard packages are required to compile and run experiments, as described in *experiments/ReadMe.md*.

Installation

There is no explicit installation step, cloning the repository is enough. The artifact is fully self-contained and does not install files to the rest of the machine, except for benchmark scripts copied to a configurable directory on the machine that runs them.

Experiment workflow

All experiments are run using scripts. Manual work is not needed beyond executing the scripts with some parameters and setting up a configuration file once.

Evaluation and expected result

The scripts produce tables and figures that correspond to those in this paper. Tables are produced as tab-separated output on the command line, while figures are produced as vector images.

Experiment customization

The benchmarking scripts are reusable for any experiment even not including TinyNF. They are designed to measure the throughput and latency of any network function, with special treatment for ones that require DPDK-compatible kernel drivers.

Notes

We elaborate here on the environment abstraction library mentioned in Section 5, which is the only dependency of the TinyNF driver. The driver itself does not depend on any kernel-mode driver and only needs "freestanding" features of the C library, i.e., it only uses a few headers and types but no C functions from the standard library.

The abstraction contains 5 groups of functions: memory allocation and deallocation, translation between virtual and physical addresses, PCI register reads and writes, endianness conversion, and sleep.

We believe these 5 groups are all necessary to write NIC drivers without compromises, though some of these could be modified or removed under certain conditions. Sleeping could be replaced by a clock function combined with busy-waiting in the driver, but this would be less efficient and no less complex. Memory deallocation could be omitted if the software uses a crash-only failure mode. Translating virtual to physical addresses may not be necessary in the presence of an IOMMU, if memory allocation also configured the IOMMU. In systems that use the Enhanced Configuration Access Mechanism for PCI registers, or "ECAM" for short, the functions to read and write PCI registers could instead be a single function providing the memory address at which this space is accessible for a given device.

Notably, the abstraction does not expose non-uniform memory access: implementations are expected to provide sane defaults. The current Linux implementation allocates memory on the same node as the current CPU and does not allow for PCI operations on devices on other nodes. This would need to change in a more production-ready version, in which various strategies can be used when dealing with devices on multiple nodes, but those strategies are beyond the scope of this paper.

AE Methodology

Submission, reviewing and badging methodology:
usenix.org/conference/osdi20/call-for-artifacts