Corundum: An Open-Source 100-Gbps NIC

Alex Forencich, Alex C. Snoeren, George Porter, George Papen
Department of Electrical and Computer Engineering
University of California, San Diego
{jforenci, snoeren, gmporter, gpapen}@eng.ucsd.edu

***Abstract*—Corundum is an open-source, FPGA-based prototyping platform for network interface development at up to 100 Gbps and beyond. The Corundum platform includes several core features to enable real-time, high-line-rate operations including: a high-performance datapath, 10G/25G/100G Ethernet MACs, PCI Express gen 3, a custom PCIe DMA engine, and native high-precision IEEE 1588 PTP timestamping. A key feature is extensible queue management that can support over 10,000 queues coupled with extensible transmit schedulers, enabling fine-grained hardware control of packet transmission. In conjunction with multiple network interfaces, multiple ports per interface, and per-port event-driven transmit scheduling, these features enable the development of advanced network interfaces, architectures, and protocols. The software interface to these hardware features is a high-performance driver for the Linux networking stack. The platform also supports scatter/gather DMA, checksum offloading, receive flow hashing, and receive-side scaling. Development and debugging is facilitated by a comprehensive open-source, Python-based simulation framework that includes the entire system from a simulation model of the driver and PCI express interface to the Ethernet interfaces. The power and flexibility of Corundum is demonstrated by the implementation of a microsecond-precision time-division multiple access (TDMA) hardware scheduler to enforce a TDMA schedule at 100 Gbps line rate with no CPU overhead.**

I. Introduction and Background

The network interface controller (NIC) is the gateway through which a computer interacts with the network. The NIC forms a bridge between the software stack and the network, and the functions of this bridge define the network interface. Both the functions of the network interface as well as the implementation of those functions are evolving rapidly. These changes have been driven by the dual requirements of increasing line rates and NIC features that support high-performance distributed computing and virtualization. The increasing line rates have led to many NIC functions that must be implemented in hardware instead of software. Concurrently, new network functions such as precise transmission control for multiple queues are needed to implement advanced protocols and network architectures.

To meet the need for an open development platform for new networking protocols and architectures at realistic line rates, we are developing an open-source[1] high-performance, FPGA-based NIC prototyping platform. This platform, called Corundum, is capable of operation up to at least 94 Gbps, is fully open source and, along with its driver, can be used across a complete network stack. The design is both portable and compact, supporting many different devices while also leaving ample resources available for further customization even on smaller devices. We show that Corundum's modular design and extensibility permit co-optimized hardware/software solutions to develop and test advanced networking applications in a realistic setting.

[1]Corundum codebase: https://github.com/ucsdsysnet/corundum

A. Motivation and previous work

The motivation for the development of Corundum can be understood by looking at how network interface features in existing NIC designs are currently partitioned between hardware and software. Hardware NIC functions fall into two main categories. The first category consists of simple offloading features that remove some per-packet processing from the CPU—such as checksum/hash computation and segmentation offloading that enables the network stack to process packets in batches. The second category consists of features that must be implemented in hardware on the NIC to achieve high performance and fairness. These features include flow steering, rate limiting, load balancing, and time stamping.

Traditionally, the hardware functions of NICs are built into proprietary application-specific integrated circuits (ASICs). Coupled with economies of scale, this enables high performance at low cost. However, the extensibility of these ASICs is limited and the development cycle to add new hardware functions can be expensive and time-consuming [1]. To overcome these limitations, a variety of smart NICs and software NICs have been developed. Smart NICs provide powerful programmability on the NIC, generally by providing a number of programmable processing cores and hardware primitives. These resources can be used to offload various application, networking, and virtualization operations from the host. However, smart NICs do not necessarily scale well to high line rates, and hardware features can be limited [1].

Software NICs offer the most flexibility by implementing network functionality in software, bypassing most of the hardware offloading features. As a result, new functions can be developed and tested quickly, but with various trade-offs including consuming host CPU cycles and not necessarily supporting operation at full line rate. Additionally, because of the inherent random interrupt-driven nature of software, the development of networking applications that require precise transmission control is infeasible [2]. Despite this, many research projects [3]–[6] have implemented novel NIC functions in software by either modifying the network stack or

by using kernel-bypass frameworks such as the Data Plane Development Kit (DPDK) [7].

FPGA-based NICs combine features of ASIC-based NICs and software NICs: they are capable of running at full-line rate and delivering low latency and precision timing, while having a relatively short development cycle for new functions. High-performance, proprietary, FPGA-based NICs have also been developed. For example, Alibaba developed a fully custom FPGA-based RDMA-only NIC that they used to run a hardware implementation of a precision congestion control protocol (HPCC) [8]. Commercial products also exist, including offerings from Exablaze [9] and Netcope [10].

Unfortunately, similar to ASIC-based NICs, commercially-available FPGA-based NICs tend to be proprietary with basic "black-box" functions that cannot be modified. The closed nature of basic NIC functionality severely limits their utility and flexibility for developing new networking applications.

Commercially-available high-performance DMA components such as the Xilinx XDMA core and QDMA cores, and the Atomic Rules Arkville DPDK acceleration core [11] do not provide fully configurable hardware to control the flow of transmit data. The Xilinx XDMA core is designed for compute offload applications and as such provides very limited queuing functionality and no simple method to control transmit scheduling. The Xilinx QDMA core and Atomic Rules Arkville DPDK acceleration core are geared towards networking applications by supporting a small number of queues and providing DPDK drivers. However, the number of queues supported is small—2K queues for the XDMA core and up to 128 queues for the Arkville core—and neither core provides a simple method for precise control over packet transmission.

Open-source projects such as NetFPGA [12] exist, but the NetFPGA project only provides a toolbox for general FPGA-based packet processing and is not specifically designed for NIC development. Moreover, the NetFPGA NIC reference design utilizes the propriety Xilinx XDMA core, which is not designed for networking applications. Replacing the Xilinx XDMA core in the reference NIC design for the NetFPGA board with Corundum results in a much more powerful and flexible prototyping platform.

FPGA based packet-processing solutions include Catapult [13], which implements network application offloading, and FlowBlaze [14], which implements reconfigurable match-action engines on FPGAs. However, these platforms leave the standard NIC functions to a separate ASIC-based NIC and operate entirely as a "bump-in-the-wire", providing no explicit control over the NIC scheduler or queues.

Other projects use software implementations or partial hardware implementations. Shoal [15] describes a network architecture that performs cell routing with custom NICs and fast Layer 1 electrical crosspoint switches. Shoal was constructed in hardware, but was only evaluated with synthetic traffic with no connection to a host. SENIC [3] describes scalable NIC-based rate-limiting. A hardware implementation of the scheduler was evaluated in isolation, but the system-level evaluation was carried out in software with a custom queuing discipline (qdisc) module. PIEO [16] describes a flexible NIC scheduler, which was evaluated in hardware in isolation. NDP [5] is a pull-mode transmission protocol for datacenter applications. NDP was evaluated with DPDK software NICs and FPGA-based switches. Loom [6] describes an efficient NIC design, which is evaluated in software with BESS.

The development of Corundum is distinguished from all of these projects because it is completely open source and can operate with a standard host network stack at practical line rates. It provides thousands of transmit queues coupled with extensible transmit schedulers for fine-grained control of flows. This leads to a powerful and flexible open-source platform for the development of networking applications that combine both hardware and software functionalities.

II. Implementation

Corundum has several unique architectural features. First, hardware queue states are stored efficiently in FPGA block RAM, enabling support for thousands of individually-controllable queues. These queues are associated with interfaces, and each interface can have multiple ports, each with its own independent transmit scheduler. This enables extremely fine-grained control over packet transmission. The scheduler module is designed to be modified or swapped out completely to implement different transmit scheduling schemes, including experimental schedulers. Coupled with PTP time synchronization, this enables time-based scheduling, including high precision TDMA.

The design of Corundum is modular and highly parametrized. Many configuration and structural options can be set at synthesis time by Verilog parameters, including interface and port counts, queue counts, memory sizes, scheduler type, etc. These design parameters are exposed in configuration registers that the driver reads to determine the NIC configuration, enabling the same driver to support many different boards and configurations without modification[2].

The current design supports PCIe DMA components for the Xilinx Ultrascale PCIe hard IP core interface. Support for the PCIe TLP interface commonly used in other FPGAs is not implemented, and is future work. This support should enable operation on a much larger set of FPGAs.

The footprint of Corundum is rather small, leaving ample space available for additional logic, even on relatively small FPGAs. For example, the Corundum design for the ExaNIC X10 [9], a dual port 10G design with a PCIe gen 3 x8 interface and 512 bit internal datapath, consumes less than a quarter of the logic resources available on the second smallest Kintex Ultrascale FPGA (KU035). Table I, placed at the end of the paper, lists the resources for several target platforms.

The rest of this section describes the implementation of Corundum on an FPGA. First, a high-level overview of the main functional blocks is presented. Then, details of several

[2]Corundum codebase: https://github.com/ucsdsysnet/corundum

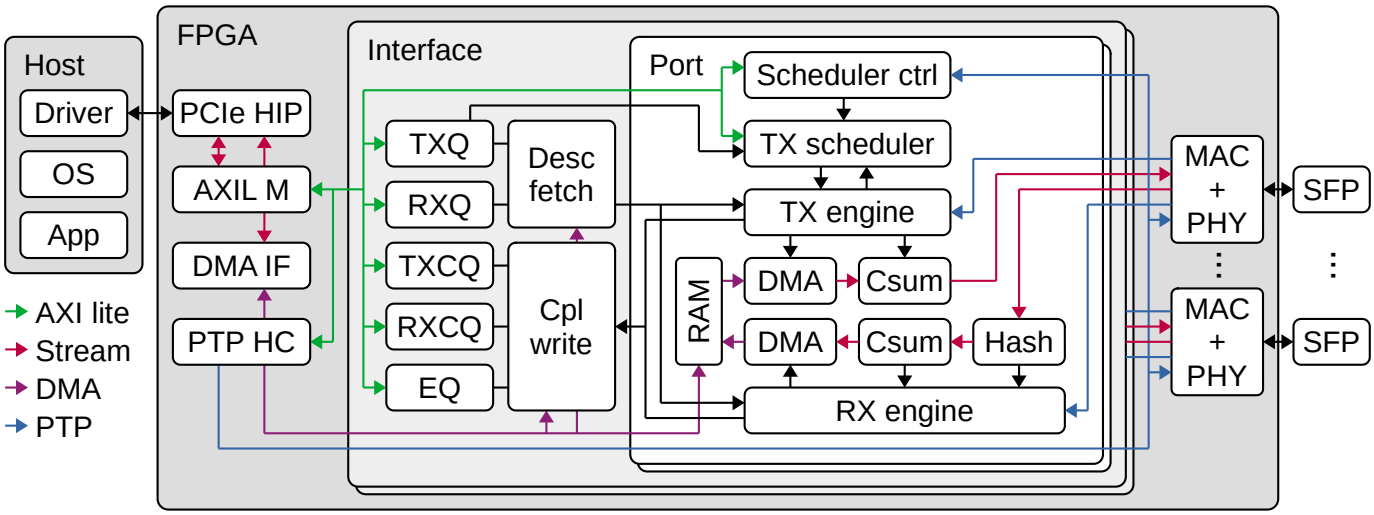


Fig. 1. Block diagram of the Corundum NIC. PCIe HIP: PCIe hard IP core; AXIL M: AXI lite master; DMA IF: DMA interface; PTP HC: PTP hardware clock; TXQ: transmit queue manager; TXCQ: transmit completion queue manager; RXQ: receive queue manager; RXCQ: receive completion queue manager; EQ: event queue manager; MAC + PHY: Ethernet media access controller (MAC) and physical interface layer (PHY).

of the unique architectural features and functional blocks are discussed.

A. High-level overview

A block diagram of the Corundum NIC is shown in Fig. 1. At a high level, the NIC consists of 3 main nested modules. The top-level module primarily contains support and interfacing components. These components include the PCI express hard IP core and DMA interface, the PTP hardware clock, and Ethernet interface components including MACs, PHYs, and associated serializers. The top-level module also includes one or more `interface` module instances. Each interface module corresponds to an operating-system-level network interface (e.g. eth0). Each interface module contains the queue management logic as well as descriptor and completion handling logic. The queue management logic maintains the queue state for all of the NIC queues—transmit, transmit completion, receive, receive completion, and event queues. Each interface module also contains one or more `port` module instances. Each port module provides an AXI stream interface to a MAC and contains a transmit scheduler, transmit and receive engines, transmit and receive datapaths, and a scratchpad RAM for temporarily storing incoming and outgoing packets during DMA operations.

For each port, the transmit scheduler in the port module decides which queues are designated for transmission. The transmit scheduler generates commands for the transmit engine, which coordinates operations on the transmit datapath. The scheduler module is a flexible functional block that can be modified or replaced to support arbitrary schedules, which may be event driven. The default implementation of the scheduler is simple round robin. All ports associated with the same interface module share the same set of transmit queues and appear as a single, unified interface to the operating system. This enables flows to be migrated between ports or load-balanced across multiple ports by changing only the transmit scheduler settings without affecting the rest of the network stack. This dynamic, scheduler-defined mapping of queues to ports is a unique feature of Corundum that can enable research into new protocols and network architectures, including parallel networks such as P-FatTree [17] and optically-switched networks such as RotorNet [18] and Opera [19].

In the receive direction, incoming packets pass through a flow hash module to determine the target receive queue and generate commands for the receive engine, which coordinates operations on the receive datapath. Because all ports in the same interface module share the same set of receive queues, incoming flows on different ports are merged together into the same set of queues. It is also possible to add customized modules to the NIC to pre-process and filter incoming packets before they traverse the PCIe bus.

The components on the NIC are interconnected with several different interfaces including AXI lite, AXI stream, and a custom segmented memory interface for DMA operations, which will be discussed later. AXI lite is used for the control path from the driver to the NIC. It is used to initialize and configure the NIC components and to control the queue pointers during transmit and receive operations. AXI stream interfaces are used for transferring packetized data within the NIC, including both PCIe transmission layer packets (TLPs) and Ethernet frames. The segmented memory interface serves to connect the PCIe DMA interface to the NIC datapath and to the descriptor and completion handling logic.

The majority of the NIC logic runs in the PCIe user clock domain, which is nominally 250 MHz for all of the current design variants. Asynchronous FIFOs are used to interface with the MACs, which run in the serializer transmit and receive clock domains as appropriate—156.25 MHz for 10G, 390.625 MHz for 25G, and 322.266 MHz for 100G.

The following sections describe several key functional

blocks within the NIC.

B. Pipelined queue management

Communication of packet data between the Corundum NIC and the driver is mediated via descriptor and completion queues. Descriptor queues form the host-to-NIC communications channel, carrying information about where individual packets are stored in system memory. Completion queues form the NIC-to-host communications channel, carrying information about completed operations and associated metadata. The descriptor and completion queues are implemented as ring buffers that reside in DMA-accessible system memory, while the NIC hardware maintains the necessary queue state information. This state information consists of a pointer to the DMA address of the ring buffer, the size of the ring buffer, the producer and consumer pointers, and a reference to the associated completion queue. The required descriptor state for each queue fits into 128 bits.

The queue management logic for the Corundum NIC must be able to efficiently store and manage the state for thousands of queues. This means that the queue state must be completely stored in block RAM (BRAM) or ultra RAM (URAM) on the FPGA. Since a 128 bit RAM is required and URAM blocks are 72x4096, storing the state for 4096 queues requires only 2 URAM instances. Utilizing URAM instances enables scaling the queue management logic to handle at least 32,768 queues per interface.

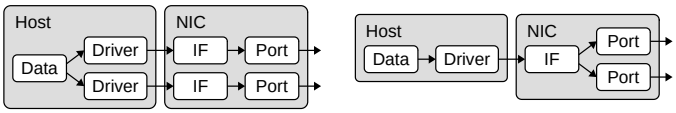
In order to support high throughput, the NIC must be able to process multiple descriptors in parallel. Therefore, the queue management logic must track multiple in-progress operations, reporting updated queue pointers to the driver as the operations are completed. The state required to track in-process operations is much smaller than the descriptor state, and as such it can be stored in flip-flops and distributed RAM.

The NIC design uses two queue manager modules: `queue_manager` is used to manage host-to-NIC descriptor queues, while `cpl_queue_manager` is used to manage NIC-to-host completion queues. The modules are similar except for a few minor differences in terms of pointer handling, fill handling, and doorbell/event generation. Because of the similarities, this section will discuss only the operation of the `queue_manager` module.

The BRAM or URAM array used to store the queue state information requires several cycles of latency for each read operation, so the `queue_manager` is built with a pipelined architecture to facilitate multiple concurrent operations. The pipeline supports four different operations: register read, register write, dequeue/enqueue request, and dequeue/enqueue commit. Register-access operations over an AXI lite interface enable the driver to initialize the queue state and provide pointers to the allocated host memory as well as access the producer and consumer pointers during normal operation.

C. Transmit scheduler

The default transmit scheduler used in the Corundum NIC is a simple round-robin scheduler implemented in the `tx_scheduler_rr` module. The scheduler sends commands to the transmit engine to initiate transmit operations out of the NIC transmit queues. The round-robin scheduler contains basic queue state for all queues, a FIFO to store currently-active queues and enforce the round-robin schedule, and an operation table to track in-process transmit operations.



(a) Traditional NIC, port assignment made in software

(b) Corundum NIC, port assignment made in hardware

Fig. 2. NIC port and interface architecture comparison

Similar to the queue management logic, the round-robin transmit scheduler also stores queue state information in BRAM or URAM on the FPGA so that it can scale to support a large number of queues. The transmit scheduler also uses a processing pipeline to hide the memory access latency.

The transmit scheduler module has four main interfaces: an AXI lite register interface and three streaming interfaces. The AXI lite interface permits the driver to change scheduler parameters and enable/disable queues. The first streaming interface provides doorbell events from the queue management logic when the driver enqueues packets for transmission. The second streaming interface carries transmit commands generated by the scheduler to the transmit engine. Each command consists of a queue index to transmit from, along with a tag for tracking in-process operations. The final streaming interface returns transmit operation status information back to the scheduler. The status information informs the scheduler of the length of the transmitted packet, or if the transmit operation failed due to an empty or disabled queue.

The transmit scheduler module can be extended or replaced to implement arbitrary scheduling algorithms. This enables Corundum to be used as a platform to evaluate experimental scheduling algorithms, including those proposed in SENIC [3], Carousel [4], PIEO [16], and Loom [6]. It is also possible to provide additional inputs to the transmit scheduler module, including feedback from the receive path, which can be used to implement new protocols and congestion control techniques such as NDP [5] and HPCC [8]. Connecting the scheduler to the PTP hardware clock can be used to support TDMA, which can be used to implement RotorNet [18], Opera [19], and other circuit-switched architectures.

D. Ports and interfaces

A unique architectural feature of Corundum is the split between the port and the network interface so that multiple ports can be associated with the same interface. Most current NICs support a single port per interface, as shown in Fig. 2a. When the network stack enqueues a packet for transmission on a network interface, the packets are injected into the network via the network port associated with that interface. However, in Corundum, multiple ports can be associated with each interface, so the decision over which port a packet will injected

into the network can be made by hardware at the time of dequeue, as shown in Fig. 2b.

All ports associated with the same network interface module share the same set of transmit queues and appear as a single, unified interface to the operating system. This enables flows to be migrated between ports or load-balanced across multiple ports by changing only the transmit scheduler settings without affecting the rest of the network stack. The dynamic, scheduler-defined mapping of queues to ports enables research into new protocols and network architectures, including parallel networks such as P-FatTree [17] and optically-switched networks such as RotorNet [18] and Opera [19].

E. Datapath, and transmit and receive engines

Corundum uses both memory-mapped and streaming interfaces in the datapath. AXI stream is used to transfer Ethernet packet data between the port DMA modules, Ethernet MACs, and the checksum and hash computation modules. AXI stream is also used to connect the PCIe hard IP core to the PCIe AXI lite master and PCIe DMA interface modules. A custom, segmented memory interface is used to connect the PCIe DMA interface module, port DMA modules, and descriptor and completion handling logic to internal scratchpad RAM.

The width of the AXI stream interfaces is determined by the required bandwidth. The core datapath logic, except the Ethernet MACs, runs entirely in the 250 MHz PCIe user clock domain. Therefore, the AXI stream interfaces to the PCIe hard IP core must match the hard core interface width—256 bits for PCIe gen 3 x8 and 512 bits for PCIe gen 3 x16. On the Ethernet side, the interface width matches the MAC interface width, unless the 250 MHz clock is too slow to provide sufficient bandwidth. For 10G Ethernet, the MAC interface is 64 bits at 156.25 MHz, which can be connected to the 250 MHz clock domain at the same width. For 25G Ethernet, the MAC interface is 64 bits at 390.625 MHz, necessitating a conversion to 128 bits to provide sufficient bandwidth at 250 MHz. For 100G Ethernet, Corundum uses the Xilinx 100G hard CMAC cores on the Ultrascale Plus FPGAs. The MAC interface is 512 bits at 322.266 MHz, which is connected to the 250 MHz clock domain at 512 bits because it needs to run at approximately 195 MHz to provide 100 Gbps.

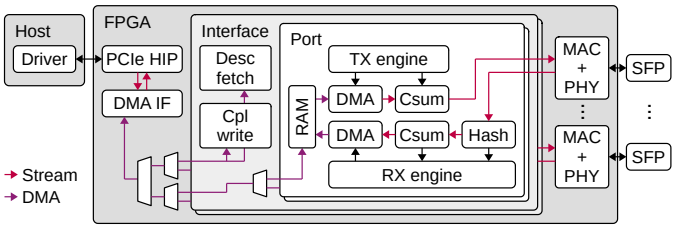


Fig. 3. Simplified version of Fig. 1 showing the NIC datapath.

A block diagram of the NIC datapath is shown in Fig. 3, which is a simplified version of Fig. 1. The PCIe hard IP core (PCIe HIP) connects the NIC to the host. Two AXI stream interfaces connect the PCIe DMA interface module to the PCIe hard IP cores. One interface for read and write requests, and one interface for read data. The PCIe DMA interface module is then connected to the descriptor fetch module, completion write module, port scratchpad RAM modules, and the RX and TX engines via a set of DMA interface multiplexers. In the direction towards the DMA interface, the multiplexers combine DMA transfer commands from multiple sources. In the opposite direction, they route transfer status responses. They also manage the segmented memory interfaces for both reads and writes. The top-level multiplexer combines descriptor traffic with packet data traffic, giving the descriptor traffic higher priority. Next, a pair of multiplexers combine traffic from multiple interface modules. Finally, an additional multiplexer inside each interface module combines packet data traffic from multiple port instances.

The transmit and receive engines are responsible for coordinating the operations necessary for transmitting and receiving packets. The transmit and receive engines can handle multiple in-progress packets for high throughput. As shown in Fig. 1, the transmit and receive engines are connected to several modules in the transmit and receive data path, including the port DMA modules and hash and checksum offload modules, as well as the descriptor and completion handling logic and the timestamping interfaces of the Ethernet MACs.

The transmit engine is responsible for coordinating packet transmit operations. The transmit engine handles transmit requests for specific queues from the transmit scheduler. After low-level processing using the PCIe DMA engine, the packet will then pass through the transmit checksum module, MAC, and PHY. Once the packet is sent, the transmit engine will receive the PTP timestamp from the MAC, build a completion record, and pass it to the completion write module.

Similar to the transmit engine, the receive engine is responsible for coordinating packet receive operations. Incoming packets pass through the PHY and MAC. After low-level processing that includes hashing and timestamping, the receive engine will issue one or more write requests to the PCIe DMA engine to write the packet data out into host memory. When the writes complete, the receive engine will build a completion record and pass it to the completion write module.

The descriptor read and completion write modules are similar in operation to the transmit and receive engines. These modules handle descriptor/completion read/write requests from the transmit and receive engines, issue enqueue/dequeue requests to the queue managers to obtain the queue element addresses in host memory, and then issue requests to the PCIe DMA interface to transfer the data. The completion write module is also responsible for handling events from the transmit and receive completion queues by enqueuing them in the proper event queue and writing out the event record.

F. Segmented memory interface

For high performance DMA over PCIe, Corundum uses a custom segmented memory interface. The interface is split into segments of maximum 128 bits, and the overall width is double that of the AXI stream interface from the PCIe hard IP core. For example, a design that uses PCIe gen

3 x16 with a 512-bit AXI stream interface from the PCIe hard core would use a 1024-bit segmented interface, split into 8 segments of 128 bits each. This interface provides an improved "impedance match" over using a single AXI interface, enabling higher PCIe link utilization by eliminating backpressure due alignment in the DMA engine and arbitration in the interconnect logic. Specifically, the interface guarantees that the DMA interface can perform a full-width, unaligned read or write on every clock cycle. Additionally, the use of simple dual port RAMs, dedicated to traffic moving in a single direction, eliminates contention between the read and write paths.

Each segment operates similar to AXI lite, except with three interfaces instead of five. One channel provides the write address and data, one channel provides the read address, and one channel provides the read data. Unlike AXI, bursts and reordering are not supported, simplifying the interface logic. Interconnect components (multiplexers) are responsible for preserving the ordering of operations, even when accessing multiple RAMs. The segments operate completely independently of each other with separate flow control connections and separate instances of interconnect ordering logic. In addition, operations are routed based on a separate select signal and not by address decoding. This feature eliminates the need to assign addresses and enables the use of parametrizable interconnect components that appropriately route operations with minimal configuration.

Byte addresses are mapped onto segmented interface addresses with the lowest-order address bits determining the byte lane in a segment, the next bits selecting the segment, and the highest-order bits determining the word address for that segment. For example, in a 1024-bit segmented interface, split into 8 segments of 128 bits, the lowest 4 address bits would determine the byte lane in a segment, the next 3 bits would determine the segment. The remainder of the bits determine the address bus for that segment.

G. Device Driver

The Corundum NIC is connected to the Linux kernel networking stack with a kernel module. The module is responsible for initializing the NIC, registering kernel interfaces, allocating DMA-accessible buffers for descriptor and completion queues, handling device interrupts, and passing network traffic between the kernel and the NIC.

The NIC uses register space to expose parameters including the number of interfaces, number of ports, number of queues, number of schedulers, maximum transport unit (MTU) size, and presence of PTP timestamping and offload support. The driver reads these registers during initialization so it can configure itself and register kernel interfaces to match the NIC design configuration. This auto-detection capability means that the driver and NIC are loosely coupled; the driver generally does not need to be modified with respect to the core datapath when used across different FPGA boards, different Corundum design variants, and different parameter settings.

H. Simulation Framework

An extensive open-source, Python-based simulation framework is included to evaluate the complete design. The framework is built using the Python library MyHDL and includes simulation models of the PCI express system infrastructure, PCI express hard IP core, NIC driver, and Ethernet interfaces. The simulation framework facilitates debugging the complete NIC design by providing visibility into the state of the entire system.

The core of the PCIe simulation framework consists of about 4,500 lines of Python and includes transaction-layer models of PCIe infrastructure components including root complex, functions, endpoints, and switches as well as high-level functionality including configuration space, capabilities, bus enumeration, root complex memory allocation, interrupts, and other functions. Additional modules, consisting of another 4,000 lines of Python, provide models of the FPGA PCIe hard IP cores, exchanging transaction-layer traffic with the simulated PCIe infrastructure and driving signals that can be connected to a cosimulated Verilog design.

Simulating Corundum requires a few lines of code to instantiate and connect all of the components. Listing 1 shows an abbreviated testbench to send and receive packets of various sizes using the simulation framework, with the Verilog design cosimulated in Icarus Verilog. The testbench instantiates simulation models for the Ethernet interface endpoints, PCIe root complex, and driver, and connects these to the cosimulated design. Then, it initializes the PCIe infrastructure, initializes the driver model, and sends, receives, and verifies several test packets of various lengths.

III. Results

The 100G variant of the Corundum NIC was evaluated on an Alpha Data ADM-PCIE-9V3 board, installed in a Dell R540 server (dual Xeon 6138), connected to a commercial state-of-the-art NIC (Mellanox ConnectX-5) in an identical server with a QSFP28 direct attach copper cable. Two more Mellanox ConnectX-5 NICs installed in the same machines were also evaluated for comparison. Up to eight instances of iperf3 were used to saturate the link.

To compare the performance of Corundum with the Mellanox ConnectX-5, both NICs were initially configured with a maximum transmission unit (MTU) of 9000 bytes. For this configuration, Corundum can separately achieve 95.5 Gbps RX and 94.4 Gbps TX (Fig. 4a). Under the same conditions, the Mellanox ConnectX-5 NIC achieves 97.8 Gbps for both RX and TX. When running additional instances of iperf to simultaneously saturate the link in both directions, the performance of Corundum degrades to 65.7 Gpbs RX and 85.9 Gbps TX (Fig. 4b). For the existing testbed, the performance of the Mellanox NIC also degraded to 83.4 Gbps for both RX and TX. The degradation of both Corundum and the ConnectX-5 in full-duplex mode suggests that the software driver may be a significant contributor to the reduction in performance. Specifically, the current version of driver only supports the Linux kernel networking stack. A reference

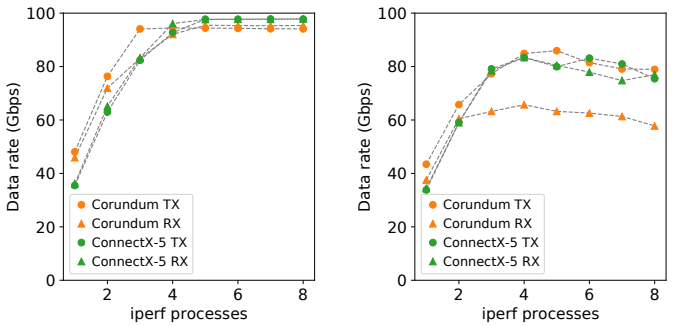
```
from myhdl import *
import pcie, pcie_us, pcie_usp, axis_ep
# signals
clk_250mhz = Signal(bool(0))
# etc.
# sources and sinks
qsfp_0_source = axis_ep.AXIStreamSource()
qsfp_0_source_logic = qsfp_0_source.create_logic(
    qsfp_0_rx_clk,
    # etc.
)
# etc.
# set up PCIe infrastructure
rc = pcie.RootComplex()
# create driver instance
driver = mqnic.Driver(rc)
# create PCIe hard IP core instance
dev = pcie_usp.UltrascalePlusPCIe()
rc.make_port().connect(dev)
pcie_logic = dev.create_logic(
    m_axis_cq_tdata=s_axis_cq_tdata,
    # etc.
)
# connect to Verilog design
dut = Cosimulation(
    "vvp -m myhdl testbench.vvp -lxt2",
    clk_250mhz=user_clk,
    # etc.
)

@instance
def check():
    # initialization
    yield rc.enumerate()
    yield from driver.init_dev(
        dev.functions[0].get_id())
    yield from driver.interfaces[0].open()
    # test packets of various lengths
    for k in range(64, 1515):
        data = bytearray([x%256 for x in range(k)])
        # send a packet
        yield from driver.interfaces[0]
            .start_xmit(data, 0)
        yield qsfp_0_sink.wait()
        pkt = qsfp_0_sink.recv()
        assert pkt.data == data
        # receive a packet
        qsfp_0_source.send(pkt)
        yield driver.interfaces[0].wait()
        pkt = driver.interfaces[0].recv()
        assert pkt.data == data
```

Listing 1. Abbreviated NIC testbench. Includes setting up PCIe, Ethernet interface, and driver models, initializing the simulated PCIe bus and driver, and sending and receiving test packets. Most signals removed for brevity.

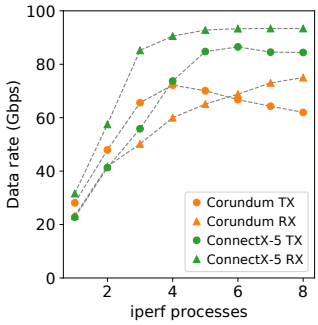
design that supports a kernel-bypass framework such as DPDK is an objective of future work. This design should improve the performance for full-duplex mode and can be customized for specific applications.

Figures 4c and 4d compare the performance for an MTU of 1500 bytes. For this case, Corundum can separately achieve 75.0 Gbps RX and 72.2 Gbps TX (Fig. 4c) and simultaneously achieve 53.0 Gbps RX and 57.6 Gbps TX (Fig. 4d). The performance difference for Corundum between TX and RX seen in Fig. 4c as the number of iperf instances increases is caused by a limitation on the number of in-process transmit packets coupled with PCIe round-trip delay. This was verified

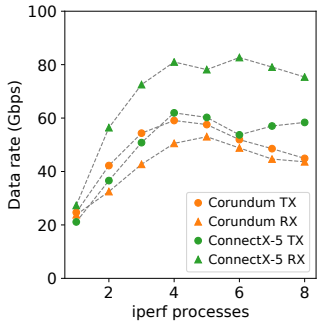


(a) 9 kB MTU, separate RX and TX.

(b) 9 kB MTU, simultaneous RX and TX.



(c) 1.5 kB MTU, separate RX and TX.



(d) 1.5 kB MTU, simultaneous RX and TX.

Fig. 4. Comparison of the TCP throughput for Corundum and a Mellanox ConnectX-5.

by increasing the number of in-process transmit operations from 8 to 16. This improved the rate from 65.6 Gbps RX and 45.7 Gbps TX to the rates 75.0 Gbps RX and 72.2 Gbps TX shown in Fig. 4c. For comparison, under the same conditions, the Mellanox ConnectX-5 NIC can separately achieve 93.4 Gbps for RX and 86.5 Gbps TX and simultaneously achieve 82.7 Gbps RX and 62.0 Gbps TX.

To test the performance of PTP timestamping, two Corundum NICs in 10G mode were connected to an Arista 40G packet switch operating as a PTP boundary clock. The NICs were configured to output a fixed frequency signal derived from PTP time, which was captured by an oscilloscope. When Corundum is implemented with PTP timestamping enabled, the hardware clocks can be synchronized with linuxptp to better than 50 ns. The time synchronization performance is unchanged when the link is saturated.

IV. Case-Study: Time-Division Multiple Access (TDMA)

Precise network admission control is a critical networking functionality at high line rates. Corundum provides thousands of transmit queues that can be used to separate and control transmit data on a fine time scale synchronized across multiple end hosts. This functionality provides a unique toolbox that can be used develop new and powerful NIC functions. Determining what network functions to implement and the impact these functions have on network performance is an active research area [3]–[5], [16].

TABLE I
RESOURCE UTILIZATION

Board	FPGA	PCIe	IF	TXQ	MTU	MAC	Speed	LUT	FF	BRAM	URAM
ADM-PCIE-9V3	XCVU3P	3 x16	2x1	8K	2 K	10 G	10 G	69.4 K (18%)	71.7 K (9%)	252 (33%)	20 (6%)
ADM-PCIE-9V3	XCVU3P	3 x16	2x1	8K	2 K	25 G	25 G	66.7 K (17%)	71.7 K (9%)	239 (33%)	20 (6%)
ADM-PCIE-9V3	XCVU3P	3 x16	2x1	8K	16 K	100 G	78.7 G	61.7 K (16%)	74.1 K (9%)	331 (33%)	20 (6%)
ExaNIC X10	XCKU035	3 x8	2x1	1K	2 K	10 G	10 G	40.7 K (20%)	45.5 K (11%)	131 (24%)	-
ExaNIC X25	XCKU3P	3 x8	2x1	8K	2 K	10 G	10 G	43.9 K (27%)	51.8 K (16%)	132 (37%)	20 (42%)
ExaNIC X25	XCKU3P	3 x8	2x1	8K	2 K	25 G	25 G	41.3 K (25%)	51.0 K (16%)	127 (35%)	20 (42%)
NetFPGA SUME	XC7V690T	3 x8	2x1	512	2 K	10 G	10 G	43.0 K (10%)	50.8 K (6%)	133 (9%)	-
VCU108	XCVU095	3 x8	1x1	2K	2 K	10 G	10 G	28.2 K (5%)	26.0 K (2%)	107 (6%)	-
VCU118	XCVU9P	3 x16	2x1	8K	2 K	10 G	10 G	70.0 K (6%)	72.5 K (3%)	252 (12%)	20 (2%)
VCU118	XCVU9P	3 x16	2x1	8K	16 K	100 G	78.7 G	62.4 K (5%)	76.8 K (3%)	331 (15%)	20 (2%)
VCU1525	XCVU9P	3 x16	2x1	8K	2 K	10 G	10 G	69.4 K (6%)	71.7 K (3%)	252 (12%)	20 (2%)
VCU1525	XCVU9P	3 x16	2x1	8K	16 K	100 G	78.7 G	62.2 K (5%)	76.1 K (3%)	331 (15%)	20 (2%)

To demonstrate how Corundum can be used for precision transmission control, we implemented a simple reference design for TDMA with a fixed schedule. Starting with this basic design and the modular structure of Corundum, custom schedulers for novel networking protocols can be implemented that are minimally intrusive to the overall architecture.

The fixed TDMA schedule can be synchronized across multiple hosts via IEEE 1588 PTP. TDMA is implemented by enabling and disabling queues in the transmit scheduler according to PTP time, under the control of the TDMA scheduler control module. Queue enable and disable commands are generated in the TDMA scheduler control module and sent to the transmit scheduler at the beginning and end of each timeslot of the TDMA schedule. The TDMA scheduler operates under the assumption that the timeslots are sufficiently long so that the TDMA scheduler control module can prepare for the next timeslot during the current timeslot. In addition, a relatively small number of queues must be active during each timeslot so the skew between the first and last queue enabled or disabled is small.

The TDMA scheduler control module runs in the 250 MHz PCIe user clock domain. As a result, it takes 4 ns per queue to iterate over each transmit queue to prepare for the next timeslot (about 32.8 us total for 8,192 transmit queues). Similarly, it takes 4 ns to generate each enable or disable command to send to the transmit scheduler module.

A. TDMA performance

The 100G TDMA variant of the Corundum NIC with 256 transmit queues was evaluated on an Alpha Data ADM-PCIE-9V3 board, installed in a Dell R540 server (dual Xeon 6138), connected to a Mellanox ConnectX-5 NIC. Eight instances of iperf3 were used to saturate the link, and both NICs were configured with an MTU of 9 kB. With TDMA disabled, the NIC runs at 94.0 Gbps. The TDMA scheduler was configured run a schedule with period 200 µs containing two timeslots of 100 µs, enabling all transmit queues in the first timeslot and disabling them in the second. Accounting for a 8 µs interval for the 11 packets in the transmit datapath (11 $\times$ 0.72 µs per packet) at 100 Gbps plus 1 µs to disable all 256 queues, Corundum could control the data leaving the NIC with a precision of two packet lengths or 1.4 µs.

An additional test was run at 10 Gbps line rate with an MTU of 1500 bytes using a schedule with a period of 200 µs. This period was partitioned into two timeslots of 100 µs. Accounting for a 38 µs interval for the 32 packets in the transmit datapath (32 $\times$ 1.2 µs per packet) at 10 Gbps plus 1 µs to disable all 256 queues, Corundum could control the data leaving the NIC with a precision of two packet lengths or 2.4 µs.

V. CONCLUSION

In this paper, we presented Corundum, an open-source, high-performance, FPGA-based NIC. The measured performance of the initial design provides realistic line rates sufficient to develop and test new networking applications. Existing and planned open-source reference designs enable customization and further performance improvements. These features lead to a powerful prototyping platform for network research and development, including NIC-based schedulers such as SENIC [3], Carousel [4], PIEO [16], and Loom [6], new protocols and congestion control techniques such as NDP [5] and HPCC [8]. Corundum also enables new parallel network architectures, such as P-FatTree [17], RotorNet [18], and Opera [19]. Optimizing the design to improve performance for smaller packet sizes as well as customizing the design for new networking protocols based on precise packet transmission are objectives of ongoing work.

REFERENCES

[1] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone

[2] B. Stephens, A. Akella, and M. M. Swift, "Your programmable NIC should be a programmable switch," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets '18. New York,

NY, USA: Association for Computing Machinery, 2018, p. 36–42. [Online]. Available: https://doi.org/10.1145/3286062.3286068

[3] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat, "SENIC: Scalable NIC for end-host rate limiting," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 475–488. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/radhakrishnan

[4] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 404–417. [Online]. Available: https://doi.org/10.1145/3098822.3098852

[5] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 29–42. [Online]. Available: https://doi.org/10.1145/3098822.3098825

[6] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 33–46. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/stephens

[7] "Data plane development kit," https://www.dpdk.org/.

[8] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and et al., "HPCC: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 44–58. [Online]. Available: https://doi.org/10.1145/3341302.3342085

[9] "Exablaze," https://exablaze.com/.

[10] "Netcope technologies," https://www.netcope.com/en.

[11] "Atomic rules," http://www.atomicrules.com/.

[12] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sep. 2014.

[13] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, and et al., "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[14] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano, "FlowBlaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 531–548. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[15] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon, "Shoal: A network architecture for disaggregated racks," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 255–270. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/shrivastav

[16] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 367–379. [Online]. Available: https://doi.org/10.1145/3341302.3342090

[17] W. M. Mellette, A. C. Snoeren, and G. Porter, "P-FatTree: A multi-channel datacenter network topology," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 78–84. [Online]. Available: https://doi.org/10.1145/3005745.3005746

[18] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter, "RotorNet: A scalable, low-complexity, optical datacenter network," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 267–280. [Online]. Available: https://doi.org/10.1145/3098822.3098838

[19] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter, "Expanding across time to deliver bandwidth efficiency and low latency," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/mellette