NanoTransport: A Low-Latency, Programmable Transport Layer for NICs

Serhat Arslan
sarslan@stanford.edu
Stanford University

Stephen Ibanez
sibanez@stanford.edu
Stanford University

Alex Mallery
amallery@stanford.edu
Stanford University

Changhoon Kim
changhoon.kim@stanford.edu
Stanford University

Nick McKeown
nickm@stanford.edu
Stanford University

ABSTRACT

Transport protocols can be implemented in NIC (Network Interface Card) hardware to increase throughput, reduce latency and free up CPU cycles. If the ideal transport protocol were known, the optimal implementation would be simple: bake it into fixed-function hardware. But transport layer protocols are still evolving, with innovative new algorithms proposed every year. A recent study proposed Tonic, a Verilog-programmable transport layer in hardware. We build on this work to propose a new programmable hardware transport layer architecture, called nanoTransport, optimized for the extremely low-latency message-based RPCs (Remote Procedure Calls) that dominate large, modern distributed data center applications. NanoTransport is programmed using the P4 language, making it easy to modify existing (or create entirely new) transport protocols in hardware. We identify common events and primitive operations, allowing for a streamlined, modular, programmable pipeline, including packetization, reassembly, timeouts and packet generation, all to be expressed by the programmer.

We evaluate our nanoTransport prototype by programming it to run the reliable message-based transport protocols NDP and Homa, as well as a hybrid variant. Our FPGA prototype – implemented in Chisel and running on the Firesim simulator – exposes P4-programmable pipelines and is designed to run in an ASIC at 200Gb/s with each packet processed end-to-end in less than 10ns (including message reassembly).

CCS CONCEPTS

• **Networks → Programming interfaces**; **Transport protocols**; • **Hardware → Networking hardware**.

ACM Reference Format:
Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. 2021. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '21), September 20–21, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3482898.3483365

1 INTRODUCTION

Modern distributed applications send huge numbers of Remote Procedure Calls (RPCs) between large groups of servers [8, 34, 62]. This has motivated new proposals to reduce RPC processing times, for example by redesigning the network interface card (NIC) to reduce the processing time on the end-host (commercial products [14, 16, 47, 48, 51, 53, 66] and research proposals [4, 19, 25, 28, 34, 36, 39–41, 43, 44, 62]), and new low-latency transport layer protocols to reduce delays caused by network congestion [3, 13, 20, 23, 24, 42, 50]. Therefore, we classify prior work to reduce latency in two main locations: (1) The *end-host*. For example eRPC [34], a software design that combines many software techniques to reduce median RPC response times to $1-2\mu$s, and NeBuLa [62], a hardware design that reduces RPC response time below 100ns by integrating the NIC with the CPU, bypassing PCIe, and placing arriving RPC requests directly into the L1 cache. (2) The *network*. For example NDP [23], which mitigates incast congestion by trimming off a packet's data in congested switches, sending only the header to the receiver, allowing it to decide when the packet should be resent.

Clearly, if we wish to minimize overall RPC response time, we need to minimize latency in the end-host *and* the network.

In this paper we focus on the transport layer. We are most interested in transport layer protocols that are low latency in two senses: Algorithms that are simple, with minimal processing time in the end-host NIC; and low latency in the sense that the algorithm minimizes congestion delays as the packet traverses the network.

Our approach is to minimize end-host latency by placing the transport layer in hardware, and to empower others to minimize congestion delays by making the hardware programmable.

Transport layer processing: Over the years, a lot of work has gone into reducing transport layer processing time in *software*. For example, Google's micro-kernel approach to host networking, Snap [45] reports an end-to-end tail latency of 100μs. Homa's Linux kernel implementation [52] can deliver an incoming message from the NIC to a user thread in about 5μs. eRPC [34], by carefully optimizing for the common case, reports 850ns wire-to-wire latency for small 32 byte RPCs with the Timely [49] congestion control protocol.

The current fastest reported combination of a whole system — a low-latency NIC with a transport layer — is the nanoPU [28], with 69ns median wire-to-wire RPC response time, using a direct message interface to the CPU register file, a hardware thread scheduler, and fixed NDP-based transport layer (with a 7ns processing time).

SOSR '21, September 20–21, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-9084-2/21/09…$15.00
https://doi.org/10.1145/3482898.3483365

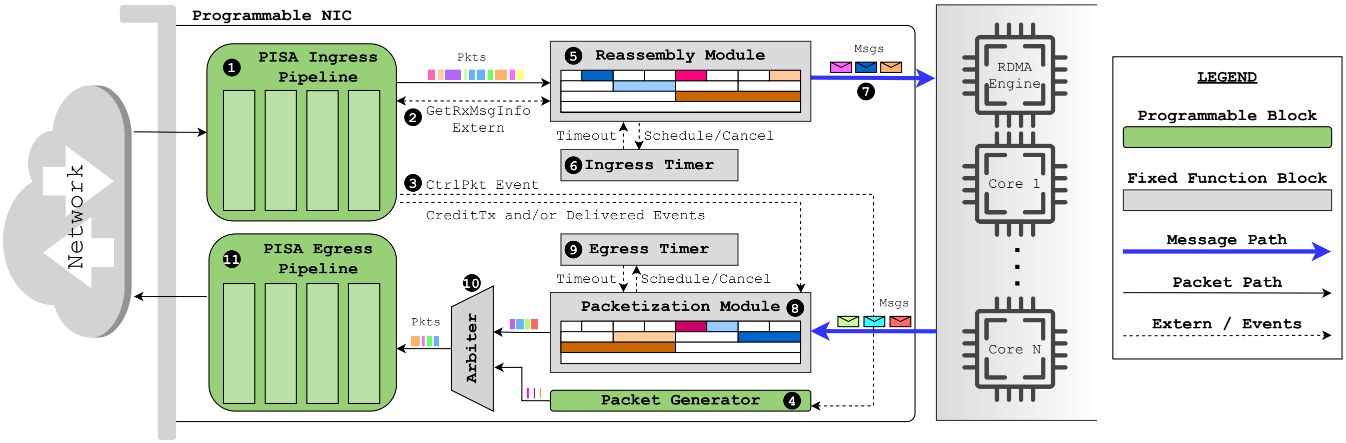


Figure 1: NanoTransport architecture design. Processing steps are numbered chronologically.

It is tempting to think that we are done: Is there much to be gained by reducing the processing path by a few more nanoseconds? The nanoPU processes packets entirely in hardware, right up until the RPC request starts processing in a thread. There appears to be little headroom to reduce latency further.

However, there is likely still room to improve latency in the network. The design of the transport layer congestion control algorithm can have big consequences on network latency. Indeed, researchers continue to propose new protocols to reduce network latency for RPC messages [1–3, 7, 13, 20, 23, 24, 50], and cloud service providers report the design and deployment of novel algorithms too [38, 42, 49, 68]. The jury is still out as to which algorithm is the best, and there may not be a single *best* algorithm; rather, it will likely depend on the particular data center topology and the specific distributed application [57]. For example, in §5.2 we show scenarios where NDP and Homa are each better than the other.

One way in which transport protocols differ is in the relative importance given to throughput and latency (median and tail). It is widely known that even if an RPC request returns a result quickly *on average*, the tail latency often dictates the application's performance, particularly if it launches multiple RPC requests to different servers and must wait for all of them to return before making progress. As the number of cascaded RPCs increase, the likelihood of a long tail latency increases, defining the end-to-end performance of the entire system [17].

The right trade-off is likely best determined by a cloud service provider. But if the transport protocol is baked into fixed function hardware, it is an expensive and time-consuming task to modify it. This observation prompted the authors of Tonic [4] to propose a *programmable* hardware design for transport protocols which "exploits the common patterns in transport logic to create reusable high-speed hardware modules." Their design assumes the transport layer will be implemented on an FPGA and that the programmer will use the Verilog [29] hardware-design language to implement a new algorithm. Because Verilog requires a steep learning curve, the authors provide an NS3 model to help users design new protocol layers. The basic model is that, once a transport connection has been established, the kernel offloads the connection state and packet processing to the NIC. The Tonic prototype utilizes ring-buffers and bitmaps to keep track of connection state and achieves 100Gb/s with 128 byte packets, and is able to process a packet in about 100ns.

While some NICs are implemented in FPGAs, ASICs (Application Specific Integrated Circuits) are trending because of their higher performance, lower power and lower cost. We set out to design a programmable transport layer inspired by Tonic, prototyped on an FPGA, but optimized for implementation in an ASIC.

We call our design nanoTransport, and it extends Tonic in several ways. First, nanoTransport is designed to run in an ASIC, and programmed (in the field) using the P4 language [11] which can achieve ∼10× faster packet processing compared to FPGAs. P4 pipelines are already used in modern commercial NICs [47, 53, 66], and an industry group is creating a standard portable architecture for P4-programmable smart NICs [18]. Second, a wide range of transport protocols share a common set of triggering events (e.g. packet arrival, timeouts, duplicate ack) and nanoTransport exploits P4's simpler and widely accepted abstractions for them. It enables interfaces to trigger events in a programmable fashion, inspired by the event-driven P4 packet processing framework [26]. Third, nanoTransport implements both the sender and the receiver clients of a transport protocol, whereas Tonic is designed to offload only the sender-side protocol. Finally, our design is streamlined: It can process packets in the transport layer in less than 10ns, issuing a new packet every 2.6ns.

NanoTransport focuses on reducing the transport related processing latency. The latency for delivering a message to the application thread after transport processing involves mechanisms such as core selection and thread scheduling which are beyond the scope of nanoTransport. Therefore, to demonstrate a complete programmable system, we prototyped nanoTransport on the open-source nanoPU design framework. [1] This allows others to experiment with our design, try out new transport layer protocols, and improve upon our work. However, our programmable transport layer is not bound to the nanoPU; it could be used as a standalone, P4-programmable pipeline in any NIC that offloads the transport

[1] We prototype our design by building upon the nanoPU RISC-V design repository [27].

layer to hardware — for example, the RDMA processing pipeline in a modern NIC.

In summary, the main contributions of nanoTransport are:

(1) We identify interfaces to a common set of events in transport protocols that can be used as primitives for a programmable solution.
(2) We observe that transport protocol processing can be efficiently expressed in the P4 programming language.
(3) We build and evaluate the first P4 programmable transport layer in hardware, that could be added to the nanoPU system, or standalone in an RDMA NIC pipeline.
(4) We provide an open source FPGA based nanoTransport prototype running at 200Gb/s, even for small packets, while maintaining less than 100 bytes of state per message.
(5) In our design, a packet can be deterministically processed in 11ns (median and tail latency), including the ingress and egress paths. This is three orders of magnitude lower than common software based implementations and an order of magnitude lower than Tonic.
(6) We provide a behavioral model of nanoTransport in NS3 [54] to help designers evaluate new transport protocols and algorithms at scale prior to programming hardware.

The remainder of the paper describes nanoTransport's building blocks in §2, design details in §3, prototype FPGA implementation in §4, and prototype evaluations in §5. We discuss use-cases, feasibility, and limitations of programmable hardware transport layers in §6.

2 TRANSPORT LAYER DISSECTED

Despite their differences, most transport protocols share a large set of features. In this section, we explore and enumerate common features that, later, we use as the basis of the nanoTransport design.

2.1 Protocol Taxonomy

Broadly speaking there are two types of transport protocols: WAN (wide area network) protocols such as TCP NewReno [21], CUBIC [22], and BBR [12]; and DC (data center) protocols, such as RoCE [46], DCQCN [68] and Timely [49].

WAN protocols are designed for long-lived, reliable bi-directional byte-streams, and the primary performance metrics are throughput and fairness. Connections are established by a handshake that installs per-connection state at both ends, and maintained for the duration of the connection. Because WAN RTTs are typically 1-100ms, a microsecond level improvement in the end-host processing does not add much value whereas our focus is on low-latency. Therefore, we will not consider this type of protocols in our design.

On the other hand, data center protocols are mostly used to exchange small messages between servers [5, 55]. RTTs are a few microseconds, and latency sensitive applications can benefit greatly from further microsecond-level reductions in the end-host processing time [23, 24, 50]. Therefore, nanoTransport focuses on *latency-sensitive, reliable, message-based transport protocols*, primarily for data centers.

Specifically, nanoTransport *is designed to allow a user to program a low-latency, reliable, one-way messaging service.*

A (beneficial) consequence of small message communication is that no persistent connection state is required. This reduces the amount of memory a NIC needs to track currently active messages, making possible faster and lower-power single-chip ASIC solutions.

2.2 Building Blocks

Our programmable hardware transport layer has two service interfaces: Below, it exchanges Ethernet frames with the Ethernet MAC. Above, it exchanges complete, reassembled, reliable messages with a CPU core or RDMA engine. Regardless of the protocol specifics, a reliable message-based transport protocol on nanoTransport must:

(1) Split an outgoing message into one or more packets. Packets are stored for retransmission until successfully delivered to the receiver.
(2) Reassemble incoming packets back into messages. Packets arriving out of order are correctly resequenced during reassembly.
(3) Maintain timers to trigger packet retransmissions or to cancel messages upon repeated failures.
(4) Maintain state for each ongoing message that can, for example, allow congestion control logic to decide which packet to send next, and when.
(5) Generate control packets to signal message state or congestion, for example, ACK, NACK, and GRANT.

A key observation of our work is that only the last two functions (maintaining per-message state, and generating control packets) require programmability to support different congestion control algorithms. The other capabilities are fixed and common to all reliable message-oriented transport protocols we have encountered.

Tonic made a similar observation [4], and elected to use bitmaps to keep track of message state and determine which packet to send next or retransmit. NanoTransport keeps these bitmaps in the reassembly and packetization modules, next to the associated packet data. Different protocols differ in how they modify the bitmaps when data or control packets are sent and received, how they detect a packet loss, and how they handle a lost packet. A nanoTransport programmer determines how events are triggered and processed (e.g.data packet arrival, packet loss detection, packet acknowledgement) via P4 `externs` [64] and by extending P4 metadata fields. §3 describes our design in more detail.

3 ARCHITECTURE DESIGN

Figure 1 shows the nanoTransport architecture. The pipeline sits between the external Ethernet packet interface (the MAC), with which it exchanges Ethernet frames, and the CPU core (or RDMA engine), with which it exchanges fully assembled, ready-to-use messages. The pipeline is self-contained and handles all aspects of the transport layer on behalf of the CPU. The CPU is needed to configure and initialize the pipeline, but, in order to minimize latency, the CPU is not involved in processing individual packets.

The design is deeply pipelined so as to process many packets in parallel (to maximize throughput), but not too deep (to keep latency low). The ingress and egress pipelines each contain a mix of fixed and programmable modules. The two pipelines also operate independently, other than when triggering a few well-defined events (described in §3.4.1 and §3.4.2).

We start by walking through the high-level steps to process arriving and departing packets and then dive deeper into each stage: An arriving packet at the NIC ❶ is first processed by the programmable ingress pipeline, where protocol-specific logic determines how the packet will be processed. The `GetRxMsgInfo` extern is then called ❷. This extern uses flow identifiers such as the 5-tuple or unique message ID to fetch (or allocate) per-message state in the reassembly module. The per-message state is common to all protocols and is described in §3.3.1. Depending on the protocol, the ingress pipeline may then also choose to trigger a `CtrlPktEvent` ❸ that causes the packet generator to generate a control packet (acknowledgement, grant or NACK etc. depending on the protocol) in response to the incoming packet ❹. The original data packet is passed to the reassembly module ❺, which stores it and checks if the message is complete. The reassembly module maintains and updates a per-message timer for incoming messages ❻. Should a timer expire (indicating message reception failure), all state for the message is garbage collected. Once all of a message's packets have been received, they are reassembled in the correct order and forwarded as a full message to the CPU (or the RDMA Engine) ❼.

In the egress direction, when a message is sent from an application thread ❽, it is stored in the packetization module, which divides the message into MTU size segments and initializes per-message state variables. A per-message retransmission timer is set ❾; if it expires, some of the message's packets may be retransmitted. When the packetization module sends a packet, it is enqueued by the arbiter ❿, which schedules its departure alongside outgoing packets from the packet generator. Finally, packets pass through the programmable egress pipeline ⓫, where protocol-specific headers are added before the packet is sent to the network.

Next, we describe each block in detail and provide the API signatures for event handling.

3.1 Programmable Components

The pipeline contains the following programmable modules: the P4 programmable PISA pipelines and the packet generator module.

PISA Pipelines. A PISA (Protocol Independent Switch Architecture) [10] pipeline provides a simple match-action abstraction, allowing fast, and flexible packet processing by executing P4 programs [11]. Our design dedicates separate PISA pipelines for ingress and egress. Each pipeline contains a standard P4 library (`core.p4`), as well as several custom externs to support nanoTransport-specific event handling logic. Users program the pipelines to parse and emit protocol-specific headers and trigger the predefined event handling logic in the fixed function blocks.

A typical ingress pipeline flow starts with a packet arriving to the parser, followed by the match tables. The tables programmed to match on protocol-specific events and this is where most protocol-specific functions are performed. For example, an ingress table may be programmed to match a flag field in the transport header; if it is a data packet, it is forwarded to the reassembly module while generating a control packet (e.g.an ACK) in response. If the incoming packet is a control packet (e.g.an ACK packet from the remote end), the ingress pipeline processes it and then discards it.

After ingress processing, data packets arrive at the reassembly module, carrying with them the metadata shown in listing 1. The metadata includes the IP address and port number of the remote sender; as well as a unique message ID (`tx_msg_id`) chosen by the sender. The three fields are used to map the message to a locally unique ID (`rx_msg_id`). The `pkt_offset` field indicates the offset of this packet within the message to which it belongs.

Listing 1: Metadata passed *from* the ingress pipeline *to* the reassembly module, along with the packet payload.

```
struct ingress_metadata_t {
    IPv4Addr_t remote_ip;
    PortNo_t   remote_port;
    bit<16>    msg_len;
    bit<8>     pkt_offset; // Similar to TCP seq no
    PortNo_t   local_port;
    MsgID_t    rx_msg_id; // Set by the receiver
    MsgID_t    tx_msg_id; // Set by the sender
    bool       is_last_pkt;
}
```

The egress pipeline's main job is to create the correct packet header for an outgoing packet. The arbiter hands the raw packet payload to the egress pipeline, which constructs the correct packet headers using the accompanying metadata. The egress metadata is shown in listing 2. The `credit` value indicates the highest packet offset that is currently authorized to be sent for this message. The `rank` is the queueing priority of the outgoing packet. For example, in Homa, the `credit` value signals which packets are granted by the receiver, and the `rank` value determines which in-network priority queue should be used by this packet. The first packet in a message has the `is_new_msg` flag set to initialize the message processing logic. The `is_rtx` flag identifies retransmitted packets, in case the protocol needs to process these packets differently.

Listing 2: Metadata passed *to* the egress pipeline along with the packet payload.

```
struct egress_metadata_t {
    IPv4Addr_t remote_ip;
    PortNo_t   remote_port;
    bit<16>    msg_len;
    bit<8>     pkt_offset;
    PortNo_t   local_port;
    MsgID_t    tx_msg_id;
    bit<16>    credit; // Similar to TCP cwnd
    bit<8>     rank; // Determines packet priority
    bit<8>     flags;
    bool       is_new_msg;
    bool       is_rtx;
}
```

In addition to header processing, transport protocols maintain protocol-specific state in the PISA pipelines. For example, NDP keeps state in the ingress pipeline to identify which packet to request in a PULL control packet. While it is a common misconception that P4 cannot be used to implement stateful logic, read-modify-write (RMW) "register" operations are frequently exposed to the programmer for stateful data plane applications in a match-action pipeline. §3.2 describes the stateful primitives in the nanoTransport PISA pipelines and §5.3 discusses the feasibility of their use.

Packet Generator. The user programs the packet generator to send protocol-specific control packets, such as NACK packets in NDP [23], GRANT packets in Homa, and INT acknowledgements in HPCC [42]. The module is triggered by `CtrlPktEvent` extern call from the ingress pipeline, which is essentially a mirrored packet carrying the metadata shown in listing 2. The metadata set by the ingress pipeline determines which control packet(s) to generate.

Different transport protocols generate control packets at different times and rates, and in different formats. For example, NDP paces its outgoing PULL control packets, used to tell the sender when to resend trimmed packets. The PULL packets must be sent at specific times. HPCC piggybacks a template to outgoing packets in the reverse direction, to carry INT reports added by switches along the path. Fortunately, the range of operations is quite small.

3.2 Stateful Primitives

This section describes the stateful primitives that can be used by the programmer in the ingress and egress PISA pipelines in order to develop protocol-specific functionality. After a survey of low-latency transport protocols, we identified a list of primitives that would be required to implement a wide range of algorithms. These primitives implement various read-modify-write (RMW) operations and are exposed to the data-plane programmer as P4 externs. Sivaraman et. al [60] propose the following set of stateful primitives that are useful across many data plane applications:

- RW – Read or write a state variable.
- RAW – Add a value to OR overwrite a state variable.
- PRAW – Perform RAW on state variable only if the provided predicate evaluates to true, otherwise leave it unchanged.
- ifElseRAW – One RAW for true and one for false predicate.

We find that for some transport protocols (e.g. NDP), these operations are sufficient. However, other protocols (e.g. Homa) require more sophisticated stateful primitives, such as multi-ported memory, to share state variables across pipeline stages, and between ingress and egress pipelines.

In addition, nanoTransport also includes a priority scheduler, which is exposed to the programmer as a P4 extern. The scheduler can store and compare multiple stateful objects using a user-provided priority value and predicate function. The programmer can insert and remove objects, and update the priority of existing objects. When called, the scheduler will return the highest priority object for which the predicate evaluates to true. §4.4 describes how we used the priority scheduler and other primitives to implement Homa's SRPT message granting logic. This primitive will be useful for other protocols as well [3, 20, 24].

According to our survey, NDP and Homa are the two protocols that together require all the identified primitives. Therefore we evaluate nanoTransport's performance on these protocols. §6.2 further discusses implementing other protocols on nanoTransport.

3.3 Reassembly Module

The reassembly module is responsible for delivering message data in the correct order. If the packets within a message arrive out of order (e.g., because of packet-by-packet multipath routing, or retransmission), the reassembly module correctly resequences them before handing the message to the application thread. Since the packet reordering logic is protocol-agnostic, nanoTransport handles it with a fixed function block.

The reassembly module maintains a bitmap for every message, called `receivedBitmap`, where each bit corresponds to a packet in the message.[2] Each packet arriving at the reassembly module is stored in the corresponding buffer. If the `is_last_pkt` flag is set on the accompanying metadata, the module forwards the entire message to the cores. The `is_last_pkt` flag is calculated during the `GetRxMsgInfo` extern call, which is described next.

3.3.1 `GetRxMsgInfo` *Extern.* The `receivedBitmap` is maintained to allow for message reassembly, and to determine which data/control packets to send next. The bitmap state is fetched by the ingress pipeline by calling the extern with the `get_rx_msg_info_req_t` metadata. The content of the input and output metadata for the `GetRxMsgInfo` extern are shown in listing 3.

Listing 3: IO for `GetRxMsgInfo` extern

```
// Metadata provided to the extern call
struct get_rx_msg_info_req_t {
  bool       mark_received; // Flag for read-only calls
  IPv4Addr_t src_ip; // Sender's IP address
  PortNo_t   src_port; // Sender's port number
  MsgID_t    tx_msg_id; // Unique ID set by the sender
  bit<16>    msg_len; // Length of the message
  bit<8>     pkt_offset; // Index of the current packet
}
// Metadata returned from the extern call
struct get_rx_msg_info_resp_t {
  bool    fail; // Extern return status
  MsgID_t rx_msg_id; // Unique ID set by the receiver
  bool    is_new_msg; // Msg not seen before
  bool    is_new_pkt; // Packet not received before
  bool    is_last_pkt; // Msg completely received
  bit<9>  ackNo; // Smallest non-received pkt_offset
}
```

The `mark_received` flag in the input metadata signals whether or not the `receivedBitmap` should be updated by the extern call. If true, the value at index `pkt_offset` is set to 1 before the output metadata is generated.[3] The remaining `get_rx_msg_info_req_t` metadata is used as the match fields of the `rx_msg_id_table` in the reassembly module, a lookup table yielding the unique locally-assigned `rx_msg_id` for the arriving message. If no entry in the table is matched, a new ID is allocated from the list of free IDs.

The `GetRxMsgInfo` extern returns `get_rx_msg_info_resp_t` metadata, including the `rx_msg_id` for the message and the state corresponding to the message. The `fail` flag signals that the reassembly module was unable to allocate resources for this message, and the programmer decides how the ingress pipeline processes such messages. `is_new_msg` is used to initialize the packet processing logic for a new message. `is_new_pkt` helps prevent processing duplicate packets. `is_last_pkt` denotes that all the bits in the `receivedBitmap` are set to 1. This value is passed along with the packet to the reassembly module, to mark message completion.

3.4 Packetization Module

NanoTransport accepts complete messages from application threads and breaks them into Ethernet packets for network transmission.

In addition to storing the message data, the packetization module maintains state variables for the message, similar to [4], so that the module can keep track of the communication between the sender and the receiver. The state variables and their roles are listed below:

[2] Every packet of a message, except the last one, is assumed to be MTU bytes long.

[3] This is useful when an arriving packet belongs to an incoming message, but it is not a data packet, e.g. trimmed packets in NDP.

- deliveredBitmap: Tracks which packets are delivered to the destination. The ingress pipeline can be programmed to trigger DeliveredEvent to update values of this bitmap. Eventually, the packetization module clears the memory allocated to the message when all of its packets are delivered.
- credit: The largest pkt_offset value that is allowed to be transmitted. All the smaller pkt_offset values are allowed to be sent into the network. The protocol logic in the ingress pipeline uses CreditTxEvent to update this value.
- txBitmap: Tracks packets that are to be (re)/transmitted. To emit a packet, the packetization module chooses the smallest index from this bitmap whose value is 1. Then, the corresponding value is reset to 0. CreditTxEvent can be triggered to set a value in this bitmap back to 1 for retransmission.
- maxTxPktOffset: Tracks the highest pkt_offset sent so far. Determines packets to be retransmitted upon a timeout.
- timeoutCnt: Tracks the number of timeouts the message has received without updating the maxTxPktOffset value. If this number is higher than the configured threshold, the packetization module gives up on the message and clears all memory allocated to it.

The packetization module also stores the message ID, sender and receiver's port numbers and receiver's IP address. The egress metadata shown in listing 2 is generated from these values whenever a packet is sent. The packetization module chooses a message from the memory which has packets that are allowed to be sent. The packet with the smallest allowed-index is forwarded to the arbiter along with the metadata.

Next, we describe the events that are used to update the packetization module's state variables.

3.4.1 DeliveredEvent.

Informs the packetization module that the packet has been successfully delivered to the remote host. The sender sets the corresponding bit in the deliveredBitmap, so that the packet is not retransmitted in the future. Typically, a received acknowledgement packet triggers this event, as decided by the programmer. Algorithm 1 shows the main processing logic triggered by this event. ackBitmap is a bitmap created by the ingress pipeline to indicate which packets to mark as delivered.

3.4.2 CreditTxEvent.

Signals that a message is currently allowed to send more packets (new packets or retransmissions). txBitmap is modified to identify which packets can be sent next time there is sufficient credit to transmit one. For example, in Homa, an arriving Grant packet triggers this event. Algorithm 2 shows the main processing logic triggered by the event. rtxBitmap is the input argument indicating which packets are to be retransmitted. It is set by the ingress pipeline under the control of the programmer. For example, NDP sets the bit for NACK packets for trimmed packets. A protocol may require several packets to be retransmitted at the same time, e.g. selective NACK similar to SACK [31].

3.4.3 TimeoutEvent.

Every message in the packetization module initiates a timer, along with metadata called rtx_offset, in the timer module. The metadata is the highest pkt_offset transmitted for the message as of the time the timer is scheduled. When a timer expires, the timer module triggers the packetization module's TimeoutEvent to compute packets for retransmission. All non-delivered packets which have smaller offset than rtx_offset are retransmitted. Finally, a new timer is scheduled for the same message to account for future retransmissions. Algorithm 3 shows the processing logic triggered by this event. Detailed description of how the timer module works is provided in §3.5.

Algorithm 1: DeliveredEvent processing logic

```
Inputs: tx_msg_id, and ackBitmap
deliveredBitmap = bitmap_table.lookup(tx_msg_id);
deliveredBitmap |= ackBitmap;
if deliveredBitmap.all() then
    ClearStateForMsg (tx_msg_id);
end
```

Algorithm 2: CreditTxEvent processing logic

```
Inputs: tx_msg_id, rtxFlag, rtxBitmap,
        creditUpdateFlag, newCredit, allowTxFlag
txBitmap = bitmap_table.lookup(tx_msg_id);
if rtxFlag then  txBitmap |= rtxBitmap ;
if creditUpdateFlag then
    currentCredit = credit_table.lookup(tx_msg_id);
    if currentCredit < newCredit then
        currentCredit = newCredit;
    end
end
// Determine which packets are allowed to be sent
txPkts = txBitmap & OnesUntil (currentCredit);
if txPkts.any() and allowTxFlag then
    Emit (txPkts);
    txBitmap &= ~txPkts;
end
```

Algorithm 3: TimeoutEvent processing logic

```
Inputs: tx_msg_id, and rtx_offset
maxTxPktOffset, timeoutCnt =
  state_table.lookup(tx_msg_id);
deliveredBitmap, txBitmap =
  bitmap_table.lookup(tx_msg_id);
if timeoutCnt > threshold then
    ClearStateForMsg (tx_msg_id);
else
    if maxTxPktOffset > rtx_offset then  timeoutCnt = 0
      else  timeoutCnt += 1 ;
    rtxPkts = (~deliveredBitmap) & OnesUntil (rtx_offset);
    if rtxPkts.any() then
        Emit (rtxPkts);
        txBitmap &= ~rtxPkts;
    end
    Timer → ScheduleEvent (tx_msg_id, maxTxPktOffset);
end
```

Our experience so far is that the fixed-function timeout event processing is sufficient for a wide class of transport protocols. However, it is possible that some protocols will need to handle timeout events differently. For example, timer events might need to generate control packets, or periodically update protocol state in the ingress/egress pipelines. Therefore, a future version of the nanoTransport architecture may benefit from making the timeout event processing programmable as well.

3.5 Timer Module

Timers are required for two purposes: (1) identify packets that have not (yet) been acknowledged and need to be retransmitted, and (2) identify messages that have been idle (have not sent or received packets) for a long time; i.e. to clean up per-message soft state.

Software implementations can maintain a timer per packet. In hardware, it is challenging to maintain a timer for every in-flight packet – potentially a large number depending on the network's BDP and the configured timeout duration. To reduce memory requirements, nanoTransport maintains a single timer *per message*.

When the applications write a new message to the packetization module, the egress timer module's ScheduleEvent is triggered. This event creates a new timer for the corresponding message, along with associated metadata. When this timer expires, the packetization module's TimeoutEvent is triggered. This event may or may not cause a new timer to be scheduled for the same message. When the message is successfully delivered to the remote client, the packetization module fires a CancelEvent within the timer module before deleting the state for the message. This event ensures that no timers are left behind which may timeout spuriously.

Similarly, when the first packet of a message arrives at the reassembly module, ScheduleEvent of the ingress timer module is triggered, which creates a new timer for the corresponding message. Since there is no notion of retransmission in the ingress direction, this timer is only used to discard the state for the message from the reassembly module. In order to prevent timeouts, each arriving message packet triggers ReScheduleEvent, which mainly resets the timer. Finally, a completed message signals CancelEvent to invalidate the associated entry in the ingress timer module.

4 HARDWARE IMPLEMENTATION

Our nanoTransport prototype extends the open source nanoPU design [28] by adding 2500 lines of Chisel [6] code and 1000 lines of P4 code. We use Firesim [35] to run large-scale, cycle-accurate simulations of our prototype on AWS FPGAs [59]. This lets us evaluate the end-to-end functionality and performance of our design. The following sections provide details of our nanoTransport prototype.

4.1 Programmable Modules

We implemented the ingress and egress pipelines using P4 and Xilinx SDNet[4] [61]. The SDNet compiler generates a Verilog module with the required functionality, which we integrate into the nanoTransport prototype. We verify correct functionality of the design using Synopsys VCS [63] cycle-accurate simulations, however, due to licensing restrictions, we are currently unable to use SDNet generated modules on AWS FPGAs. As a result, we hand-compiled our P4 code into Chisel so that we can evaluate the full system with Firesim on AWS FPGAs. The evaluation results described in §5 use our hand-compiled P4 code. Each P4 program is implemented as a custom pipeline, similar to how SDNet maps P4 programs to FPGAs. An ASIC prototype would instead have a fixed number of pipeline stages which all programs must be mapped to; we plan to explore this approach in future work.

Recall that the Packet Generator in nanoTransport is a programmable module. When processing a CtrlPktEvent from the ingress Pipeline, the packet generator might generate one or more control packets while (optionally) pacing their transmission rate. We observe that these operations are not particularly well-suited for a P4 pipeline, which is typically used to transform individual packets. As a result, in our current prototype we program the packet generator in Chisel. We will explore more convenient, higher level abstractions for programmable packet generation in future work. One possibility is to use P4 along with new custom externs to fork (duplicate) and pace packets within the pipeline.

4.2 Reassembly and Packetization Modules

The reassembly module reassembles packets, which might arrive out-of-order, into contiguous messages for delivery to applications. The packetization module splits messages into segments, which might get retransmitted out-of-order due to packet loss in the network. In order for these tasks to be performed at line rate, we must use simple data structures which require only constant time operations. We could choose from several different approaches; this section describes the buffer design in our prototype.

Our message buffer is divided into buffers of several different fixed sizes, and a free list for each size class keeps track of which buffers are available. When a buffer is to be allocated, the smallest available one that is large enough to store the whole message is selected. For message reassembly, a buffer is allocated when the first packet of the message arrives from the network and is freed when the message is forwarded to the processing cores.[5] For message packetization, a buffer is allocated when the application writes the first word of the message and is freed when the entire message has been successfully delivered to the receiver. The design uses a table indexed by message identifier to keep track of where each message is stored (the buffer pointer).

One of the benefits of using fixed size buffers to store messages is that it simplifies out-of-order reassembly and retransmission: to find the position of a particular packet within the message, the hardware simply adds the appropriate offset to the message's buffer pointer. In addition, the logic that is required to manage memory buffers is very simple and can run at line rate. Buffer allocation requires one dequeue from a free list, and freeing a buffer requires one enqueue to a free list; there is no need for complex partitioning and merging of variable size buffers.

The primary drawback of using fixed size buffers is that it leads to memory fragmentation and potentially poor utilization of the buffer space. It is therefore important to properly configure these message buffer modules. Configuration involves selecting how to carve the total buffer space into fixed size buffers. If the message

[4] Xilinx SDNet is also known as Vitis Networking P4.

[5] An arriving packet is dropped at the ingress of the reassembly module if it is unable to allocate a buffer for the message.

size distribution is known at configuration time, then it is often possible to achieve very high utilization of the buffer.

4.3 Timer Modules

The timer modules in the nanoTransport architecture maintain a single timer, along with associated metadata, for each message in the reassembly / packetization modules. Our aim is to minimize memory and logic requirements while ensuring that timers can be scheduled or canceled in constant time. Furthermore, since timers are used to trigger packet retransmissions or for garbage collection in the background, we do not need the timers to expire exactly on time, nor do we need them to expire in the correct order. The main requirement is that they expire within a bounded amount of time.

These requirements lead to a very simple hardware design. The timers for each message are stored in a single memory indexed by message ID. The entry contains the following fields: a single valid bit indicating whether or not the entry is valid, a 64-bit timeout value indicating the time at which the timer expires, and associated timer metadata. A background thread sequentially scans the entries and checks if the timer has expired. If so, it will extract the metadata and trigger a timeout event. Scheduling and canceling a timer simply involves writing a single entry to memory. In some cases, a timer may expire immediately after the background thread checks it, in which case the timeout event will not be triggered until the background thread loops back around to it. However, note that even in this case the timer will expire within a bounded amount of time, which is determined by the maximum number of timers/messages in the system. This simple design meets our requirements.

4.4 Protocol Implementations

To evaluate nanoTransport we program it to support two different protocols, chosen to represent a relatively wide range of features required by other protocols [2, 3, 7, 13, 20, 24].

NDP [23] is the first protocol we programmed on our prototype. NDP is receiver driven and aims to reduce the tail latency of network messages by ensuring that all dropped packets are retransmitted quickly. When congested, NDP-enabled switches trim data packets that would otherwise be dropped, forwarding only the packet headers to the receiver, at high priority. The receiver then quickly sends negative acknowledgements (NACKs) to inform the sender of the packet loss. This mechanism allows NDP to avoid relying on long timeouts. NDP senders initially send only up to one bandwidth-delay-product (BDP) worth of packets; the receiver explicitly pulls the remaining ones, while pacing them to ensure that the arrival rate of the pulled packets does not exceed the capacity of the bottleneck link. New data packets are pulled round-robin among messages, with the assumption that if a data packet leaves the network, a new one can be inserted without overwhelming it.

Algorithm 4 provides pseudocode for our NDP implementation in P4. The protocol uses a stateful operation to read the previous credit for the message and increment it if needed. This operation is represented with the `IfElseRaw` extern, described in §3.2.

Algorithm 4: NDP P4 Pseudocode

```
Control Ingress(out ingress_metadata):
    state credit;
    if hdr.ndp.flags.DATA then
        msg_info = GetRxMsgInfo();
        if hdr.ndp.flags.TRIM then
            genNACK = true; pull_offset_diff = 0; Drop();
        else
            genACK = true; pull_offset_diff = 1
        end
        if !msg_info.fail && msg_info.is_new_pkt then
            // ifElseRAW extern
            if msg_info.is_new_msg then
                credit[msg_info.id] = ... // initialize
            else
                credit[msg_info.id] += pull_offset_diff;
            end
            pull_offset = credit[msg_info.id];
            CtrlPktEvent(genACK, genNACK, pull_offset);
        end
    else
        if hdr.ndp.flags.ACK then DeliveredEvent();
        if hdr.ndp.flags.NACK || hdr.ndp.flags.PULL then CreditTxEvent();
        Drop();
    end
Control Egress(in egress_metadata):
    hdr.ethernet.SetValid();
    hdr.ip.SetValid();
    hdr.ndp.SetValid();
    FillHeadersFromMetaData(egress_metadata);
```

Homa [50] is the second protocol we programmed on our prototype. Homa is also a receiver driven protocol, but unlike NDP, it is designed with the assumption that packet loss is extremely rare in modern networks. Thus, it simply relies on timeouts to detect dropped packets rather than utilizing packet trimming within the network. However, it does require switches to support at least a few strict priority queues. Additionally, rather than using a round robin "pull" mechanism, Homa aims to minimize message completion time by approximating SRPT [58] scheduling at the receiver. We use the priority scheduler extern described in §3.2 to implement Homa's SRPT message granting logic. The scheduler maintains metadata about all active messages, and we assign the rank (i.e.priority) to be the remaining size of the message (lower value is higher priority). The scheduler returns the highest priority "grantable" message, where a grantable message is one that has fewer than one BDP of data outstanding. Messages are removed from the scheduler after they have been fully granted.

The implementation of the priority scheduler takes advantage of the fact that most messages are small (less than 1 BDP) and hence do not need to be scheduled; only a few messages need to be scheduled at any given time. Therefore, the scheduler extern maintains the message state in registers so that it can compare them all simultaneously. Our prototype scheduler extern supports up to 16 scheduled messages for simultaneous comparison whereas the

Algorithm 5: Homa P4 Pseudocode

```
state msgPrio;
Control Ingress(out ingress_metadata):
    state msgState;
    priorityScheduler grantScheduler;
    if hdr.homa.flags.DATA then
        msg_info = GetRxMsgInfo();
        if !msg_info.fail && msg_info.is_new_pkt then
            // ifElseRAW extern
            if msg_info.is_new_msg then
                msgState[msg_info.id] = ... // initialize
            else
                msgState[msg_info.id].remaining_size -= 1;
            end
            sched_msg = grantScheduler.apply(...);
            if sched_msg then
                // RAW extern
                msgState[sched_msg.id].grantedIdx =
                  sched_msg.grant_offset;
                CtrlPktEvent(msgState[sched_msg.id]);
            end
        end
    else
        DeliveredEvent();
        if hdr.homa.flags.GRANT then
            msgPrio[hdr.homa.tx_msg_id] = hdr.homa.prio;
            CreditTxEvent();
        end
        Drop();
    end
Control Egress(in egress_metadata):
    hdr.ethernet.SetValid();
    hdr.ip.SetValid();
    // RW extern
    hdr.ip.tos = msgPrio[egress_metadata.tx_msg_id];
    hdr.homa.SetValid();
    FillHeadersFromMetaData(egress_metadata);
```

remaining scheduled messages, if any, are stored in a FIFO queue until a register space opens.

In addition to the scheduler, Homa uses two dual ported memory primitives (§3.2) as shown in Algorithm 5. One of those dual ported memories is used to maintain information about messages – it is accessed/updated by data packets as they arrive, then updated further down the pipeline after deciding which message to grant. The other dual ported memory is used to track the priority of messages being transmitted. Incoming GRANT packets update the memory and outgoing data packets read it. Hence this state is shared between the ingress and egress pipelines.

To evaluate the programmability of our prototype, we created a new low-latency, reliable message transport protocol that we call ***Homa-Tr***. Homa-Tr combines features from NDP and Homa, in the manner a user programmer might pick and choose features from different protocols. We chose to include NDP's ability to quickly recover from packet loss by trimming packets in the switches and sending negative acknowledgements (NACKs). We adopt Homa's ability to reduce message completion time by GRANT'ing messages in SRPT order. It proved relatively quick and easy to implement Homa-Tr, incorporating NDP's packet trimming and NACK mechanism into Homa. Evaluation details are provided in §5.2.

P4 source code for the protocols is available in our open source artifact [27]. Our NDP and Homa implementations required 376 and 520 lines of P4 code, respectively, which is an order order of magnitude less code than available software based implementations.

5 EVALUATION

We evaluate the performance, correctness, and feasibility of our transport protocol implementations on the nanoTransport architecture. To evaluate performance and correctness, we run microbenchmarks and end-to-end experiments using cycle-accurate simulations on AWS FPGAs [59] with Firesim [35]. The FPGAs run at 90MHz and we simulate a target CPU and NIC clock rate of 3.2GHz. All of the results reported in this section are based upon the target 3.2GHz clock rate. To evaluate the feasibility of deploying our design in hardware, we examine the FPGA resource utilization and compare to a more traditional, open source NIC, called IceNIC [35], which does not implement the transport layer in hardware.

The design, implementation, and testing cycle for hardware prototyping is slow and expensive (even on FPGAs). Yet transport protocol designers generally need to conduct large-scale experiments to verify a protocol's functionality and usefulness. In order to ease the development process, we also developed a C++ based behavioral model for the nanoTransport architecture in NS3 [54]. A protocol is first tested at scale using NS3, before programming the hardware. Since the performance results are the same for our NS3 model and the hardware prototype, we omit them here. The source code for the NS3 behavioral model is provided as a part of the open source artifact along with the hardware prototype [27].

5.1 Latency and Throughput Microbenchmarks

NanoTransport is designed to process packets at 200Gb/s. For 1088 byte packets, 200Gb/s means that a new packet can be transmitted or received every 44ns. We verify that this is the case in the incast experiment described in §5.2.

We also evaluate the maximum throughput for the worst case traffic pattern. To measure the RX throughput, we send small 65 byte packets at 200Gb/s (380Mrps) to a nanoTransport receiver. Each packet is a separate message and carries 1 byte of payload as well as 64 bytes of packet header. We verify that the minimum sized incoming messages are forwarded to the cores at line rate. On the TX side, we generate the same workload on cores and verify that the outgoing messages are transmitted at line rate onto the wire. Our prototype is able to support the target throughput of 200Gb/s in the worst case for both NDP and Homa implementations.

Table 1 shows the RX and TX latency breakdown for our NDP and Homa implementations. Homa's ingress and egress pipelines utilize five and two stages respectively and have a slightly higher latency due to its central message scheduling decisions, whereas NDP

Table 1: RX and TX latency (first byte in to first byte out) on our nanoTransport architecture for the NDP and Homa implementations when processing a single 16 byte message (80 byte packet).

	RX Latency (ns)			TX Latency (ns)			Grand
	Ingress	Reassembly	Total	Packetize	Egress	Total	Total (ns)
NDP	5	0.94	5.94	2.81	0.31	3.12	9.06
Homa	6.25	0.94	7.19	2.81	0.94	3.75	10.94

utilizes only three and one stage. The number of stages is determined by sequential dependencies between extern calls / memory accesses. Nevertheless, the transport processing requires at most 7.2ns in the ingress path, and 3.8ns in the egress path, resulting in a maximum transport layer round-trip time of 11ns.

NanoTransport's latency is three orders of magnitude lower than the 4.8μs reported for the Homa Linux Kernel Module [52]. The latency through the Linux network stack is very sensitive to interrupt processing overheads and OS thread scheduling decisions. Ousterhout [52] reports *tail* round-trip latency of 15.1μs, 23.4μs, and 24.1μs for Homa, TCP and DCTCP respectively.

eRPC [34] is a state-of-the-art, low-latency software network stack and reports a wire-to-wire latency of 850ns. It is difficult to compare nanoTransport directly to eRPC's transport layer. However, the paper does report measurements which suggest that the congestion control logic adds an average of 17.8ns of per-packet software latency, which is comparable to nanoTransport's latency. That being said, this measurement is reported under best case conditions in which the network is not congested and thus most of the congestion control logic is bypassed for almost all packets. On the other hand, the nanoTransport latency values reported in Table 1 are deterministic. Furthermore, the eRPC measurement does not include other aspects of the transport protocol such as message packetization/reassembly or retransmission logic. Finally, as a result of running in software, a single eRPC core can only process up to about 10Mrps, which is about 38× lower throughput than the pipelined nanoTransport design.

5.2 End-to-end Evaluation

In order to evaluate the end-to-end performance, functionality of the architecture and protocol implementations (i.e. NDP, Homa, and Homa-Tr) we ran incast experiments using Firesim. In these experiments, ten senders each transmit one message to the same receiver at the same time. Each message has a distinct size, ranging from 20 to 38 MTU sized (1088B) packets. This experiment is run on a simple dumbbell topology; the bottleneck link is the receiver's down link. The RTT between the sender and the receiver is 525ns, and all the links run at 200Gb/s. We run two experiments, one in which the bottleneck buffer size is large enough to absorb the incast; and one in which the bottleneck buffer size is too small to absorb the incast, resulting in packet loss and/or trimming.

We verify the correctness of the protocol programs by examining the packet traces of the incast. Figure 2a shows the bottleneck queue occupancy in each experiment. As expected, the NDP client PULLs a data packet every time it receives one, so that the total number of packets in flight, and hence the queue occupancy, stays high until some messages complete. On the other hand, the Homa client sends GRANTs for only a few messages,[6] which allows the queue occupancy to stabilize at a low level after the first RTT of the incast.

Figure 2b shows the message completion time slowdown for each message in each of the two experiments. We define slowdown as the ratio of actual message completion time to the ideal completion time without any congestion in the network (smaller is better).

When the buffer is large, packets are not lost, enabling both NDP and Homa to smoothly PULL/GRANT new packets from the senders. However, Homa achieves lower slowdowns because messages are GRANT'ed in SRPT order – a policy designed to minimize message completion time. Since larger messages wait until the smaller ones complete, the slowdown for Homa increases with the message size. On the other hand, NDP pulls messages in a round-robin fashion, causing similarly high slowdowns across all messages.

When the buffer size is too small to absorb the incast, the relative performance of the protocols completely changes. In this case, NDP is able to achieve lower slowdowns because it enables senders to quickly retransmit lost data using packet trimming and NACKs. Homa, on the other hand, relies on timeouts to detect packet loss. Therefore, NDP still achieves similar slowdowns for all the messages, whereas it takes longer for Homa to complete the messages.

We programmed our nanoTransport prototype to implement a new protocol called Homa-Tr (§4.4) which combines features of Homa and NDP. Homa-Tr incorporates NDP's packet trimming and NACK'ing mechanism into Homa so that messages are granted in SRPT order while enabling quick recovery from packet loss. Figure 2b shows that Homa-Tr performs exactly the same as Homa when the buffer size is sufficiently large. However, when the buffer size is reduced by half (54KB), Homa-Tr is able to quickly recover from losses, and achieve ~2× better slowdown compared to Homa and ~1.5× better slowdown compared to NDP.

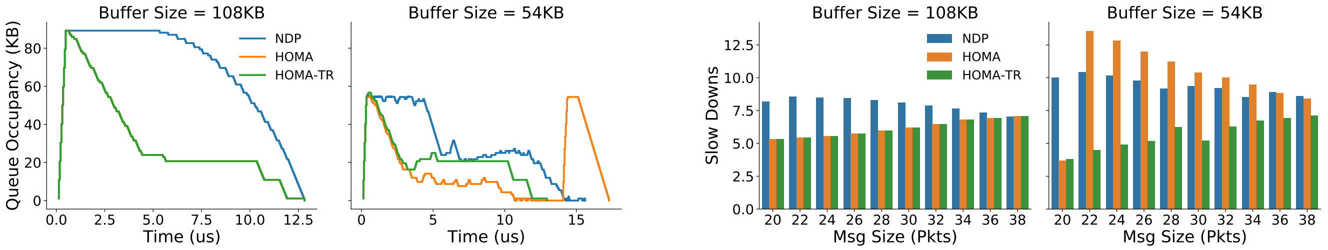
Experimental results suggest that the nanoTransport architecture can be programmed to run different low-latency protocols, and that our protocol implementations behave as expected.

5.3 Feasibility

We evaluate the cost of implementing a programmable transport layer in hardware. Figure 3 shows the FPGA resource utilization for our NDP and Homa implementations. To gauge the cost of putting the transport layer in hardware, we compare the resources used by nanoTransport against a baseline, called IceNIC [35], which does not implement any transport processing.

A basic NIC, like IceNIC, is very simple: It contains Ethernet header parsing, some staging memory and the DMA logic to transfer packets to and from host memory. Relative to IceNIC, nanoTransport adds all the transport logic described above, and Figure 3 shows

[6] Homa sends GRANTs to multiple messages, called overcommitment, to account for cases where some senders are busy with sending other messages.



(a) Bottleneck queue occupancy

(b) Message completion slowdowns.

Figure 2: Ten incast messages to the same receiver with different transport protocols and bottleneck buffer sizes. Sender and receiver NICs are all running the nanoTransport prototype.

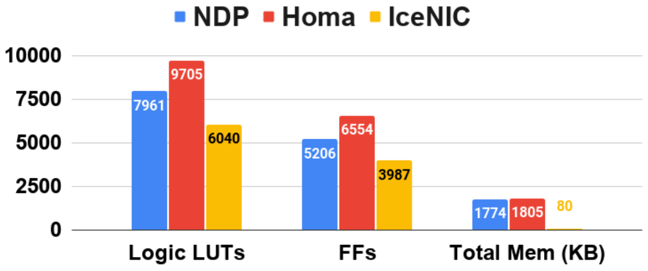


Figure 3: FPGA resource utilization of nanoTransport when running NDP and Homa (39KB max message size and 16 concurrent messages) compared to traditional IceNIC, which does not implement any transport processing.

that the logic and flip flop utilization grows by about 30% for NDP and 60% for Homa. This is as much a reflection of the simplicity of the simple IceNIC as the additional complexity of nanoTransport: Table 2 shows that nanoTransport consumes less than 2% of the logic and flip-flops of a Virtex Ultrascale+ FPGA [67]; it would require a much smaller fraction of an ASIC.

NanoTransport also requires memory for packetization and reassembly as opposed to IceNIC. The amount of memory depends on the number of concurrent messages. When designed to support up to 16 concurrent 39kB messages, nanoTransport needs about 1.2MB of on-chip SRAM[7] (Table 2). If instead 128 concurrent 39kB messages are supported, it consumes 8.4MB, which occupies less than 2mm^2 on a modern 7nm ASIC. The memory requirement increases linearly with the number of concurrent messages supported.

We conclude that nanoTransport could easily be added to a modern NIC. Modern NIC ASICs already include tens of MB of onboard SRAM [66]; adding the logic and memory for a programmable, low-latency, reliable messaging transport layer appears to be a relatively small additional cost.

6 DISCUSSION

6.1 FPGA versus ASIC

Our prototype was built to run on an FPGA as a proof-of-concept and evaluation platform. Yet our design is not necessarily the right choice for an FPGA-based NIC, where the FPGA itself can be reoptimized for a new transport protocol using Verilog.

However, ASICs mostly run faster, consume less power and cost less in volume than FPGAs [15]. NanoTransport, while tested on FPGA, is designed to be implemented in a custom NIC ASIC. In future work, we plan to synthesize the nanoTransport design and develop an ASIC implementation, possibly with a RISC-V CPU core.

Table 2: The resource utilization of our NDP prototype when configured to support both 16 and 128 concurrent 32KB messages. The percentage in each entry indicates the % utilization of the corresponding resource available on the Virtex Ultrascale+ FPGA.

# Msgs	Logic LUTs	Flip Flops	Total Mem (MB)
16	6999 (0.59%)	5043 (0.21%)	1.2 (11.9%)
128	23578 (1.99%)	8941 (0.38%)	8.4 (85.7%)

6.2 Programming New Protocols

So far, we programmed and evaluated nanoTransport when running low-latency receiver-driven protocols, NDP, Homa and Homa-Tr. For comparison, we also evaluated what it would take to program nanoTransport to run the HPCC [42], which is sender-driven (rather than receiver-driven). An HPCC sender examines the stack of INT reports [37] in every packet, determines the bottleneck link, and calculates the new window size. The PISA pipelines in nanoTransport can be used to process INT reports given switches are capable of generating them. If needed, P4 programmable switches can leverage the optimizations proposed in PINT [9] to reduce the amount of processing, thus pipeline stages, in the NIC.[8] The sender nanoTransport client can then use simple lookup tables in the P4 pipeline to calculate the congestion window size.

We also evaluated implementing DCQCN [68] and Swift [38], both of which require floating point computation at the NIC to calculate rates and congestion windows. Our nanoTransport prototype does not support floating point operations. This leaves three design choices: (1) Add floating point to the P4 pipeline in hardware; assuming we need about 200 million floating point operations per second, this is relatively straightforward in a modern ASIC, (2) Use

[7] Including message payload and the associated state, as described in §3.4.

[8] The switches compute the link utilization along the path instead of the end host.

higher precision fixed point arithmetic, which is already supported in switch ASICs [30], or (3) Use lookup tables in the P4 pipeline. We anticipate ASIC implementations will utilize all three techniques.

6.3 Multiple Concurrent Protocols

The CPU might host multiple applications, each requiring high performance transport protocols; hence the NIC may need to support several protocols at the same time. For example, it might offer a tail latency-optimized protocol for RPCs, while running a throughput-optimized protocol for the same application's bulk transfer traffic.

NanoTransport can do this, provided it has sufficient resources. Essentially, the programmable parser branches depending on the transport protocol identifier in the packet header, and the corresponding control logic is applied.

Care would need to be taken by the protocol designer to avoid undesirable interaction between the different transport protocols in the network. This is not specific to nanoTransport; it is a problem that all cloud service providers need to solve, whether the transport layer is in hardware or software. For example, Homa and NDP both assume that its receiver is the only entity allocating bottleneck bandwidth to the incoming messages. The PULL/GRANT mechanism of Homa and NDP may over/under-utilize the bottleneck link if the link is shared with non-GRANTed/PULLed traffic.

6.4 Encryption and Compression

Network operators may choose to use encrypted traffic in their network for security reasons. Modern NICs commonly include dedicated hardware modules for end-to-end encryption, and to compress data to and from storage [47, 51, 53]. Although we did not include such modules in our prototype, an ASIC implementation of nanoTransport could easily include them in its processing pipeline.

6.5 Serializing RPC Data

Low-latency reliable message protocols frequently carry RPC requests, which need to be serialized and deserialized at each end. It was recently observed that this process can add quite a lot of latency to RPC requests [65]. Zerializer shows how marshalling and unmarshalling can be done in hardware. While beyond the scope of this paper, we would anticipate ASIC implementations of nanoTransport to add such capabilities to the hardware P4 pipeline.

6.6 Scalability

A key design choice when designing a nanoTransport ASIC will be the size of the SRAM. Once picked at design time, all programmed protocols will need to live within the constraint. This means the ASIC designer needs to decide, up front, how many messages can be supported, and the size of the largest message. Our prototype supported up to 128 concurrent 32kB messages, which is reasonable for Homa and NDP. However, a more careful study of other transport protocols is needed before committing the size to an ASIC.

Careful consideration is also required when choosing the number of P4 pipeline stages, which in turn determines how many serially-dependent operations can be performed on each packet header. Our NDP and Homa programs require significantly fewer stages than the 10–20 stages commonly supported in commercial P4 pipelines today; however, more protocols should be evaluated before committing to an ASIC design.

6.7 Other Use-Cases

In addition to RPCs, small messages are frequently sent for RDMA operations as well. Typically, an RDMA-enabled NIC terminates transport logic with a fixed protocol, i.e. RoCEv2 or Infiniband, and directly accesses host memory without bothering the host CPU. NanoTransport can do the same by sending reassembled messages directly to the DMA engine. We anticipate this approach would be commonly supported on ASIC implementations of nanoTransport.

Moreover, an ASIC design would likely be configurable to bypass the packetization and reassembly module, for transport layers that application developer prefers to process in software. This would be especially useful for applications which implement complex transport features that are not available on hardware.

The available PISA pipelines also enable running data plane programs that are not transport layer related, such as NetCache [33], SwitchML [56], and PPS [32] as long as enough TCAM, SRAM, and pipeline stages are available. We leave exploration of other services that can be offloaded onto nanoTransport as future work.

7 CONCLUSION

The slowing of Moore's Law and Dennard Scaling means single core performance is leveling off; and new applications must be distributed across an ever increasing number of cores. This is helped by steadily increasing network speeds. Server NICs have transitioned quickly from 10Gb/s to 25Gb/s, 100Gb/s and now 400Gb/s.

But all too often the benefits of "many cores and a fast network" are lost because of an inefficient NIC design, or software in the network stack, or a sub-optimal network congestion control algorithm.

It is therefore natural to consider offloading transport layer into pipelined NIC hardware which runs at line rate with very low latency. But despite decades of research, the community has yet to identify a single protocol that performs the best for every edge case – there is no one size fits all. Flexibility to program the stack and deploy tailored protocols is vital, at least for the time being.

The key takeaway from nanoTransport is that it is possible to build a very high throughput (200Gb/s) NIC, with transport layer that is very low latency (10ns round-trip), yet is programmable. Our design exposes a one-way reliable message delivery interface to supply ready-to-use messages to CPU cores or an RDMA engine which further helps accelerating network stack performance.

We are still in the early days of cloud computing. Cloud service providers and their customers are still learning how to develop large, and fast distributed applications that perform well on a shared infrastructure. As they learn more, they will likely want to invent and try out new transport layer protocols. Our work demonstrates that this is possible, without compromising throughput or latency.

ACKNOWLEDGMENTS

We thank our shepherd Feng Qian, and Muhammad Shahbaz for their invaluable feedback. This work was supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-20-C-0107 and FA8650-18-2-7865.

REFERENCES

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 63–74. https://doi.org/10.1145/1851275.1851192

[2] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 19.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. PFabric: Minimal near-Optimal Datacenter Transport. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 435–446. https://doi.org/10.1145/2534169.2486031

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 93–109. https://www.usenix.org/conference/nsdi20/presentation/arashloo

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, IEEE Press, New York, NY, USA, 1212–1221. https://doi.org/10.1145/2228360.2228584

[7] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2017. PIAS: Practical Information-Agnostic Flow Scheduling for Commodity Data Centers. *IEEE/ACM Trans. Netw.* 25, 4 (Aug. 2017), 1954–1967. https://doi.org/10.1109/TNET.2017.2669216

[8] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54. https://doi.org/10.1145/3015146

[9] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 662–680. https://doi.org/10.1145/3387514.3405894

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[11] Mihai Budiu and Chris Dodd. 2017. The P416 Programming Language. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 5–14. https://doi.org/10.1145/3139645.3139648

[12] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-based Congestion Control. *Commun. ACM* 60, 2 (Jan. 2017), 58–66. https://doi.org/10.1145/3009824

[13] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/3098822.3098840

[14] Cisco. 2021. *Nexus SmartNIC*. Cisco Systems, Inc. https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html

[15] Intel Corporation. 2021. Comparing FPGAs, Structured ASICs, and Cell-Based ASICs. https://www.intel.com/content/www/us/en/products/programmable/fpga-vs-structured-asic.html. Accessed on 2021-05-09.

[16] Alpha Data. 2019. *ADM-PCIE-9V3*. Alpha Data Parallel Systems. https://www.alpha-data.com/pdfs/adm-pcie-9v3.pdf

[17] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[18] Andy Fingerhut, Radostin Stoyanov, and Nate Foster. 2021. *Portable NIC Architecture*. P4 Language Consortium. https://github.com/p4lang/pna/blob/main/generated-html/PNA-v0.5.0.pdf

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[20] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (Heidelberg, Germany) *(CoNEXT '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/2716281.2836086

[21] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. 2012. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582. https://doi.org/10.17487/RFC6582

[22] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[23] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. 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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/3098822.3098825

[24] Shuihai Hu, Wei Bai, Baochen Qiao, Kai Chen, and Kun Tan. 2018. Augmenting Proactive Congestion Control with Aeolus. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking* (Beijing, China) *(APNet '18)*. Association for Computing Machinery, New York, NY, USA, 22–28. https://doi.org/10.1145/3232565.3232567

[25] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (Princeton, NJ, USA) *(HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 60–68. https://doi.org/10.1145/3365609.3365856

[26] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. 2019. Event-Driven Packet Processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (Princeton, NJ, USA) *(HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 133–140. https://doi.org/10.1145/3365609.3365848

[27] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, and Muhammad Shahbaz. 2020. *nanoPU GitHub*. Stanford University. https://github.com/l-nic

[28] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Nick McKeown, and Changhoon Kim. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi21/presentation/ibanez

[29] IEEE (Ed.). 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

[30] Intel. 2021. *Tofino 2: Second-generation P4-programmable Ethernet switch ASIC that continues to deliver programmability without compromise.* Intel Corporation. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[31] Van Jacobson and Robert Braden. 1988. TCP extensions for long-delay paths. RFC 1072. https://doi.org/10.17487/RFC1072

[32] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast String Searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '19)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/3314148.3314356

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[34] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[35] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, IEEE Press, New York, NY, USA, 29–42.

[36] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[37] Changhoon Kim, Parag Bhide, Ed Doe, Hugh Holbrook, Anoop Ghanwani, Dan Daly, Mukesh Hira, and Bruce Davie. 2016. *Inband Network Telemetry (INT)*. P4.org. https://p4.org/assets/INT-current-spec.pdf

[38] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 514–528. https://doi.org/10.1145/3387514.3406591

[39] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. *Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs*. Association for Computing Machinery, New York, NY, USA, 36–51. https://doi-org.stanford.idm.oclc.org/10.1145/3445814.3446696

[40] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[41] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2934872.2934897

[42] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 44–58. https://doi.org/10.1145/3341302.3342085

[43] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Boston, MA, 243–259. https://www.usenix.org/conference/osdi20/presentation/lin

[44] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3341302.3342079

[45] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657

[46] Mellanox. 2014. *RoCE in the Data Center*. Mellanox Technologies. Retrieved May 31, 2021 from https://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf

[47] Mellanox. 2021. *BlueField SmartNIC for Ethernet - High Performance Ethernet Network Adapter Cards*. Mellanox Technologies. https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf

[48] Mellanox. 2021. *Mellanox Innova-2 Flex Open Programmable SmartNIC*. Mellanox Technologies. https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf

[49] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 537–550. https://doi.org/10.1145/2785956.2787510

[50] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 221–235. https://doi.org/10.1145/3230543.3230564

[51] Netronome. 2021. *About Agilio SmartNICs*. Netronome. https://www.netronome.com/products/smartnic/overview/

[52] John Ousterhout. 2021. A Linux Kernel Implementation of the Homa Transport Protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Boston, MA, 99–115. https://www.usenix.org/conference/atc21/presentation/ousterhout

[53] Pensando. 2021. *DSC-100 Distributed Services Card*. Pensando Systems. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-100-Product-Brief.pdf

[54] George F. Riley and Thomas R. Henderson. 2010. *The ns-3 Network Simulator*. Springer Berlin Heidelberg, Berlin, Heidelberg, 15–34. https://doi.org/10.1007/978-3-642-12331-3_2

[55] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 123–137. https://doi.org/10.1145/2785956.2787472

[56] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Boston, MA, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

[57] Michael Schapira and Keith Winstein. 2017. Congestion-Control Throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (Palo Alto, CA, USA) *(HotNets-XVI)*. Association for Computing Machinery, New York, NY, USA, 122–128. https://doi.org/10.1145/3152434.3152446

[58] Linus E. Schrage and Louis W. Miller. 1966. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research* 14, 4 (1966), 670–684. https://doi.org/10.1287/opre.14.4.670 arXiv:https://doi.org/10.1287/opre.14.4.670

[59] Amazon Web Services. 2006. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/. Accessed on 2020-08-10.

[60] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

[61] Henning Stubbe. 2017. P4 compiler & interpreter: A survey. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)* 7 (May 2017), 47–52. https://doi.org/10.2313/NET-2017-05-1_07

[62] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. 2020. The NeBuLa RPC-Optimized Architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, New York, NY, USA, 199–212. https://doi.org/10.1109/ISCA45697.2020.00027

[63] Synopsys. 2021. *VCS: Industry's Highest Performance Simulation Solution*. Synopsys, Inc. https://www.synopsys.com/verification/simulation/vcs.html

[64] Han Wang, Andy Fingerhut, Santiago Bautista, Nate Foster, and Chris Dodd. 2021. *Portable Switch Architecture*. P4 Language Consortium. https://p4lang.github.io/p4-spec/docs/PSA.pdf

[65] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Virtual) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3458336.3465283

[66] Xilinx. 2021. *Alveo Adaptable Accelerator Cards for Data Center Workloads*. Xilinx. https://www.xilinx.com/products/boards-and-kits/alveo.html

[67] Xilinx. 2021. *Virtex Ultrascale+ FPGA*. Xilinx. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

[68] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484