Enabling Programmable Transport Protocols in High-Speed NICs

Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, David Wentzlaff

Abstract

Data-center network stacks are moving into hardware to achieve 100 Gbps data rates and beyond at low latency and low CPU utilization. However, hardwiring the network stack in the NIC would stifle innovation in transport protocols. In this paper, we enable programmable transport protocols in high-speed NICs by designing Tonic, a flexible hardware architecture for transport logic. At 100 Gbps, transport protocols must generate a data segment every few nanoseconds using only a few kilobits of per-flow state on the NIC. By identifying common patterns across transport logic of different transport protocols, we design an efficient hardware "template" for transport logic that satisfies these constraints while being programmable with a simple API. Experiments with our FPGA-based prototype show that Tonic can support the transport logic of a wide range of protocols and meet timing for 100 Gbps of back-to-back 128-byte packets. That is, every 10 ns, our prototype generates the address of a data segment for one of more than a thousand active flows for a downstream DMA pipeline to fetch and transmit a packet.

1 Introduction

Transport protocols, along with the rest of the network stack, traditionally run in software. Despite efforts to improve their performance and efficiency [1,6,23,30], software network stacks tend to consume 30-40% of CPU cycles to keep up with applications in today's data centers [23,30,36].

As data centers move to 100 Gbps Ethernet, the CPU utilization of software network stacks becomes increasingly prohibitive. As a result, multiple vendors have developed hardware network stacks that run entirely on the network interface card (NIC) [8,10]. However, there are only two main transport protocols implemented on these NICs, both hardwired and modifiable only by the vendors:

RoCE. RoCE is used for Remote Direct Memory Access (RDMA) [8], using DCQCN [41] for congestion control and a simple go-back-N method for reliable data delivery.

TCP. A few vendors offload a TCP variant of their choice to the NIC to either be used directly through the socket API (TCP Offload Engine [10]) or to enable RDMA (iWARP [7]).

These protocols, however, only use a small fixed set out of the myriad of possible algorithms for reliable delivery [15, 19, 22, 25, 31, 32] and congestion control [11, 16, 17, 33, 40, 41] proposed over the past few decades. For instance, recent work suggests that low-latency data-center networks can significantly benefit from receiver-driven transport protocols [19,22,34], which is not an option in today's hardware stacks. In an attempt to deploy RoCE NICs in Microsoft data centers, operators needed to modify the data delivery algorithm to avoid livelocks in their network but had to rely on the NIC vendor to make that change [20]. Other algorithms have been proposed to improve RoCE's simple reliable delivery algorithm [29,32]. The long list of optimizations in TCP from years of deployment in various networks is a testament to the need for programmability in transport protocols.

In this paper, we investigate *how to make hardware transport protocols programmable*. Even if NIC vendors open up interfaces for programming their hardware, it takes a significant amount of expertise, time, and effort to implement transport protocols in high-speed hardware. To keep up with 100 Gbps, the transport protocol should generate and transmit a packet *every few nanoseconds*. It should handle *more than a thousand active flows*, typical in today's data-center servers [14, 35, 36]. To make matters worse, NICs are *extremely constrained* in terms of the amount of their on-chip memory and computing resources [28, 32].

We argue that transport protocols on high-speed NICs can be made programmable *without* exposing users to the full complexity of programming for high-speed hardware. Our argument is grounded in two main observations:

First, programmable *transport logic* is the key to enabling flexible hardware transport protocols. An implementation of a transport protocol performs several functionality such as connection management, data buffer management, and data transfer. However, its central responsibility, where most of the innovation happens, is to decide which data segments to transfer (data delivery) and when (conges-

tion control), which we collectively call the *transport logic*. Thus, the key to programmable transport protocols on high-speed NICs is enabling users to modify the transport logic.

Second, we can exploit common patterns in transport logic to create reusable high-speed hardware modules. Despite their differences in application-level API (e.g., sockets and byte-stream abstractions for TCP vs. the message-based Verbs API for RDMA), and in connection and data buffer management, transport protocols share several common patterns. For instance, different transport protocols use different algorithms to detect lost packets. However, once a packet is declared lost, reliable transport protocols prioritize its retransmission over sending a new data segment. As another example, in congestion control, given the parameters determined by the control loop (e.g., congestion window and rate), there are only a few common ways to calculate how many bytes a flow can transmit at any time. This enables us to design an efficient "template" for transport logic in hardware that can be programmed with a simple API.

Using these insights, we design and develop Tonic, a programmable hardware architecture that can realize the *transport logic* of a broad range of transport protocols, using a simple API, while supporting 100 Gbps data-rates. Every clock cycle, Tonic generates the address of the next segment for transmission. The data segment is fetched from memory by a downstream DMA pipeline and turned into a full packet by the rest of the hardware network stack (Figure 1).

We envision that Tonic would reside on the NIC, replacing the hard-coded transport logic in hardware implementations of transport protocols (e.g., future RDMA NICs and TCP offload engines). Tonic provides a unified programmable architecture for transport logic, independent of how specific implementations of different transport protocols perform connection and data buffer management, and their application-level APIs. We will, however, describe how Tonic interfaces with the rest of the transport layer in general (§2) and how it can be integrated into Linux Kernel to interact with applications using socket API as an example (§5).

We implement a Tonic prototype in ∼8K lines of Verilog code and demonstrate Tonic's programmability by implementing the transport logic of a variety of transport protocols [12, 15, 21, 22, 32, 41] in less than 200 lines of code. We also show, using an FPGA, that Tonic meets timing for ∼100 Mpps, i.e., supporting 100Gbps of back-to-back 128B packets. That is, every 10ns, Tonic can generate the transport metadata required for a downstream DMA pipeline to fetch and send one packet. From generation to transmission, the latency of a single segment address through Tonic is $\sim 0.1\mu s$, and Tonic can support up to 2048 concurrent flows.

2 Tonic as the Transport Logic

This section is an overview of how Tonic fits into the transport layer (§2.1), and how it overcomes the challenges of implementing transport logic on high-speed NICs (§2.2).

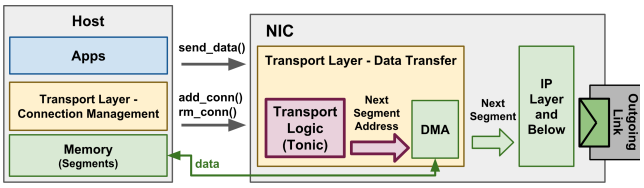

Figure 1: Tonic providing programmable transport logic in a hardware network stack on the NIC (sender-side).

2.1 How Tonic Fits in the Transport Layer

Sitting between applications and the rest of the stack, transport-layer protocols perform two main functions:

Connection Management includes creating and configuring endpoints (e.g., sockets for TCP and queue-pairs for RDMA) and establishing the connection in the beginning, and closing the connection and releasing its resources at the end.

Data Transfer involves delivering data from one endpoint to another, reliably and efficiently, in a stream of segments [1]. Different transport protocols provide different APIs for applications to request data transfer: TCP offers the abstraction of a byte-stream to which applications can continuously append data, while in RDMA, each "send" call to a queue-pair creates a separate work request and is treated as a separate message. Moreover, specifics of managing applications' data buffers differ across different implementations of transport protocols. Regardless, the transport protocol must deliver the outstanding data to its destination in multiple data segments that fit into individual packets. *Deciding which bytes comprise the next segment and when it is transmitted* is done by data delivery and congestion control algorithms, which we collectively call **transport logic** and implement in Tonic.

Figure 1 shows a high-level overview of how Tonic fits in a hardware network stack. To decouple Tonic from specifics of connection management and application-level APIs, connection setup and tear-down run outside of Tonic. Tonic relies on the rest of the transport layer to provide it with a unique identifier (*flow id*) for each established connection, and to explicitly add and remove connections using these IDs.

For data transfer on the sender side, Tonic keeps track of the number of outstanding bytes and transport-specific metadata to implement the transport logic, i.e., *generate the address of the next data segment* for each flow at the time designated by the congestion control algorithm. Thus, Tonic does not need to store and/or handle actual data bytes; it relies on the rest of the transport layer to manage data buffers on the host, DMA the segment whose address is generated in Tonic from memory, and notify it of new requests for data transmission on existing connections (see §5 for details).

The receiver-side of transport logic mainly involves gen-

[1] We focus on reliable transport as it is more commonly used and more complicated to implement.

erating control signals such as acknowledgments, per-packet grant tokens [19, 22, 34], or periodic congestion notification packets (CNPs) [41], while the rest of the transport layer manages receive data buffers and delivers the received data to applications. While handling received data can get quite complicated, generating control signals on the receiver is typically simpler than the sender. Thus, although we mainly focus on the sender, we reuse modules from the sender to implement a receiver solely for generating per-packet cumulative and selective acks and grant tokens at line rate.

2.2 Hardware Design Challenges

Implementing transport logic at line rate in the NIC is challenging due to two main constraints:

Timing constraints. Data centers have a median packet size of less than 200 bytes [14, 35]. To achieve 100 Gbps for these small packets, the NIC has to send a packet every $\sim$10 ns. Thus, the transport logic should determine the next segment for transmission every $\sim$10 ns. To perform complex operations under this timing constraint, we could conceivably pipeline the processing of transport events (e.g., segment generation, acknowledgments, timeouts) across multiple stages. However, processing back-to-back events for the same flow requires updates to the same state, making it difficult to pipeline event processing while providing state consistency. Thus, we strive to process concurrent transport events within 10 ns instead, so that we can quickly consolidate the state for the next event.

Memory constraints. A typical data-center server has more than a thousand concurrent active flows with kilobytes of in-flight data [14, 35, 36]. Since NICs have just a few megabytes of high-speed memory [28,32], the transport protocol can store only a few kilobits of state per flow on NIC.

Tonic's goal is to satisfy these tight timing and memory constraints while supporting programmability with a simple API. To do so, we identify common patterns across transport logic in various protocols that we implement as reusable fixed-function modules. These patterns allow us to optimize these modules for timing and memory, while simplifying the programming API by reducing the functionality users must specify. These patterns are summarized in Table 1, and are discussed in detail in next section, where we describe Tonic's components and how these patterns affect their design.

3 Tonic Architecture

Transport logic at the sender is what determines, for each flow, which data segments to transfer (data delivery) and when (congestion control). Conceptually, congestion control algorithms perform *credit management*, i.e., determine how many bytes a given flow can transmit at a time. Data delivery algorithms perform *segment selection*, i.e., decide which contiguous sequence of bytes a particular flow should transmit. Although the terms "data delivery" and "congestion control" are commonly associated with TCP-based transport protocols, Tonic provides a general programmable architecture for transport logic that can be used for other kinds of transport protocols as well, such as receiver-driven [19, 22, 34] and RDMA-based [8] transport protocols.

#	Observation	Examples
1	Only track a limited window of segments	TCP, NDP, IRN
2	Only keep a few bits of state per segment	TCP, NDP, IRN, RoCEv2
3	Lost segments first, new segments next	TCP, NDP, IRN, RoCEv2
4	Loss detection: Acks and timeouts	TCP, NDP, IRN
5	The three common credit calculation patterns: window, rate, and grant tokens	TCP, RoCEv2, NDP

Table 1: Common transport logic patterns.

Tonic exploits the natural functional separation between data delivery and credit management to partition them into two components with separate state (Figure 2). The data delivery engine processes events related to generating, tracking, and delivery of segments, while the credit engine processes events related to adjusting each flow's credit and sending out segments addresses for those with sufficient credit.

At the cost of lightweight coordination between the two engines, this partitioning helps Tonic meet its timing constraints while concurrently processing multiple events (e.g., receipt of acknowledgments and segment transmission) every cycle. These events must read the current state of their corresponding flow, update it, and write it back to memory for events in the next cycle. However, concurrent read and write to memory in every cycle is costly. Instead of using a wide memory to serve all the transport events, the partitioning allows the data delivery and credit engines to have narrower memories to serve only the events that matter for their specific functionality, hence meeting timing constraints.

First, §3.1 presents how the two engines coordinate to fairly and efficiently pick one of thousand flows every cycle for segment transmission while keeping the outgoing link utilized. Next, §3.2 and §3.3 describe fixed-function and programmable event processing modules in each engine, and how their design is inspired by patterns in Table1, using TCP-based, receiver-driven, and RDMA-based protocols as examples. We present Tonic's solution for resolving conflicts when more than one event for the same flow is received in a cycle in §3.4, and its programming interface in §3.5.

3.1 Efficient Flow Scheduling

At any time, a flow can only transmit a data segment if it (1) has enough credit, and (2) has a new or lost segment to send. To be work conserving, Tonic must track the set of flows that are eligible for transmission (meet both of the above criteria) and only pick among those when selecting a flow for transmission each cycle. This is challenging to do efficiently. We have more than a thousand flows with their state partitioned across two engines: Only the credit engine knows how much credit a flow has, and only the data delivery engine knows the status of a flow's segments and can

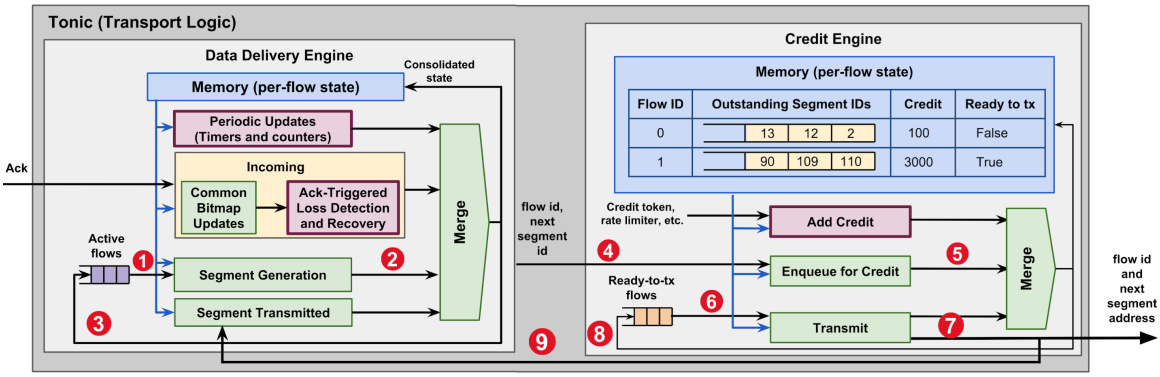

Figure 2: Tonic's architecture (red boxes (also with thick borders) are programmable, others are fixed)

generate the address of its next segment. We cannot check the state of all the flows every cycle across both engines to find the ones eligible for transmission in that cycle.

Instead, we decouple the *generation* of segment addresses from their *final transmission* to the DMA pipeline. We allow the data delivery engine to generate up to N segment addresses for a flow without necessarily having enough credit to send them out. In the credit engine, we keep a ring buffer of size N for each flow to store these outstanding segments addresses. When the flow has enough credit to send a segment, the credit engine dequeues and outputs a segment address from the buffer and signals the data delivery engine to decrement the number of outstanding segments for that flow.

This solves the problem of the partitioned state across the two engines. The data delivery engine does not need to keep track of the credit changes of flows for segment address generation. It only needs to be notified when a segment address is dequeued from the buffer. Moreover, the credit engine does not need to know the exact status of all flow's segments. If the flow's ring buffer is empty, that flow does not have segments to send. Otherwise, there are already segment addresses that can be output when the flow has enough credit.

Still, the data delivery engine cannot simply check the state of all the flows every cycle to determine those that can generate segments. Instead, we dynamically maintain the set of *active* flows in the data delivery engine, i.e., the flows that have at least one segment to generate and less than N outstanding segments (see red numbered circles in Figure 2). When a flow is created, it is added to the active set. Every cycle, one flow is selected and removed from the set for segment generation (Step 1). Once processed (Step 2), only if it has more segments to send and less than N outstanding segments, is it inserted back into the set (Step 3). Otherwise, it will be inserted in the set if, later on, the receipt of an ack or a signal from the credit engine "activates" the flow (Step 9). Moreover, the generated segment address is forwarded to the credit engine (Step 4) for insertion in the ring buffer (Step 5).

Similarly, the credit engine maintains the set of *ready-to-transmit* flows, i.e., the flows with one segment address or more in their ring buffers and enough credit to send at least one segment out. Every cycle, a flow is selected from the set (Step 6), one segment address from its ring buffer is transmitted (Step 7), its credit is decreased, and it is inserted back into the set if it has more segment addresses and credit for further transmission (Step 8). It also signals the data delivery engine about the transmission (Step 9) to decrement the number of outstanding segments for that flow.

To be fair when picking flows from the active (or ready-to-transmit) set, Tonic uses a FIFO to implement round-robin scheduling among flows in the set (see active list in [37]). The choice of round-robin scheduling is not fundamental; any other scheduler that meets our timing constraints can replace the FIFO to support other scheduling disciplines [38].

3.2 Flexible Segment Selection

With B bytes of credit, a flow can send $S = max(B, MSS)$ bytes, where MSS is the maximum segment size. In transport protocols, data-delivery algorithms use acknowledgments to keep track of the status of each byte of data (e.g., delivered, lost, in-flight, and not transmitted), and use that to decide which contiguous S bytes of data to transmit next.

However, there are two main challenges in implementing data-delivery algorithms in high-speed NICs. First, due to memory constraints, the NIC cannot store per-byte information. Second, with a few exceptions [8, 32], these algorithms are designed for software, where they could store and freely loop through large arrays of metadata to aggregate information. This computational flexibility has created significant diversity across these algorithms. Unfortunately, NIC hardware is much more constrained than software. Thus, we did not aim to support all data-delivery algorithms. Instead, we looked for patterns that are common across a variety of algorithms while being amenable to hardware implementation.

3.2.1 Pre-Calculated Fixed Segment Boundaries

Data-delivery algorithms could conceivably choose the next S bytes to send from anywhere in the data stream and produce segments with variable boundaries. However, since the NIC cannot maintain per-byte state, Tonic requires data to be partitioned into fixed-size segments (by a Kernel mod-

ule or the driver, see §5) when the flow requests transmission of new data. This way, data-delivery algorithms can use *per-segment* information to select the next segment.

With message-based transport protocols (e.g., RoCEv2), having fixed segment boundaries fits naturally; the message length is known from the beginning and can be optimally partitioned into segments. For transport protocols that provide a byte-stream abstraction (e.g., TCP and NDP), having fixed segment boundaries does not affect high-bandwidth flows as their data can be partitioned into MSS-sized segments. For flows that generate small data segments and sporadically, there is a possibility of creating many small segments, and they do not benefit much from Tonic (see §5). Regardless, due to memory constraints, segmentation is done outside of Tonic and does not affect high-bandwidth flows.

3.2.2 Small Per-Segment State for a Limited Window

Independent of a flow's available credit, data-delivery algorithms typically do not transmit a new segment if it is too far, i.e., more then C segments apart, from the first unacknowledged segment, to limit the state that the sender and receiver need to keep [2]. Still, in a 100 Gbps network with a 10µs RTT, C can get as large as ∼128 segments. Fortunately, we observe that storing the following per-segment state is enough for most data-delivery algorithms: (1) Is the segment acknowledged (in presence of selective acknowledgments)? (2) If not, is it lost or still in flight? (3) If lost, is it already retransmitted (to avoid redundant retransmission)?

More specifically, we observe that in the absence of explicit negative acknowledgments, data-delivery algorithms accumulate evidence of loss for each segment from positive acknowledgments, e.g., duplicate cumulative (e.g., TCP NewReno [21]) or selective acknowledgments (e.g., IRN for RDMA and TCP SACK [15]). Once the accumulated evidence for a segment passes a threshold, the algorithm can declare it lost with high confidence. Typically, an evidence of loss for segment i is also an evidence of loss for every *unacknowledged* segment j with $j < i$. As a result, most of these algorithms can be rewritten to only keep track of the total evidence of loss for the first unacknowledged segment and incrementally compute the evidence for the rest as needed.

Based on these observations (#1 and #2 in Table 1), we use a fixed set of bitmaps in Tonic's data delivery engine to track the status of a flow's segments and implement optimized fixed-function bitmap operations for updating them on various transport events.

3.2.3 Concurrent Event Processing

For every flow, four main events can affect the generation of its next segment address. First, the receipt of an acknowledgment can either move the window forward and enable the flow to generate more segments, or signal segment loss and trigger retransmissions. Second, the absence of acknowledgments, i.e., a timeout, can also lead to more segments marked as lost and trigger retransmissions. Third, generation of a segment address increments the number of a flow's outstanding segments and can deactivate the flow if it goes above N. Fourth, segment address transmission (out of the credit engine) decrements the number of outstanding segments and can enable the flow to generate more segment addresses.

Tonic's data delivery engine has four modules to handle these four events (Figure 2). Every cycle, each module reads the state of the flow for which it received an event from the memory in the data delivery engine, processes the event, and updates the flow state accordingly. The flow state in the data delivery engine consists of a fixed set of variables to track the status of the current window of segments across events, as well as the user-defined variables used in the programmable components (Table 2). As an example of the fixed state variables, Tonic keeps a fixed set of bitmaps for each flow (observations in §3.2.2): The `acked` bitmap keeps track of selectively acknowledged segments, `marked-for-rtx` keeps track of lost segments that require retransmission, and `rtx-cnt` stores information about their previous retransmissions.

The following paragraphs briefly describe how each event-processing module affects a flow's state, and whether there are common patterns that we can exploit to implement all or parts of its functionality in a fixed-function manner. For programmable modules, the detailed API is covered in §3.5.

Incoming. This module processes acknowledgments (and other incoming packets, see §3.3.3). Some updates to state variables in response to acknowledgments are similar across all data-delivery algorithms and do not need to be programmable (e.g., updating window boundaries, and marking selectively acked segments in `acked` bitmap, see §3.2.2), whereas loss detection and recovery, which rely on acknowledgments as a signal, vary a lot across different algorithms and must be programmable by users (#4 in Table 1). Thus, the Incoming module is designed as a two-stage pipeline: a fixed-function stage for the common updates followed by a programmable stage for loss detection and recovery.

The benefit of this two-stage design is that the common updates mostly involve bitmaps and arrays (§3.2.2), which are implemented as ring buffers in hardware and costly to modify across their elements. For instance, in all data delivery algorithms, if an incoming packet acknowledges segment C cumulatively and segment S selectively, `wnd-start` is updated to `max(wnd-start, C)` and `acked[S]` to one, and the boundaries of all bitmaps and arrays are updated based on the new `wnd-start`. By moving these updates into a fixed function stage, we can (i) optimize them to meet Tonic's timing and memory constraints, and (ii) provide the programmers with a dedicated stage, i.e., a separate cycle, to do loss detection and recovery, where they can use the updated state variables from the previous stage, the rest of the variables

[2] In TCP-based protocols, C is the minimum of receive window and congestion window size. However, the limit imposed by C exists when transport protocols use other ways (e.g., rate) to limit a flow's transmission pace [8].

State Variable	Description
acked	selectively acknowledged segments (bitmap)
marked-for-rtx	lost segments marked for retransmission (bitmap)
rtx-cnt	number of retransmissions of a segment (bitmap)
wnd-start	the address of the first segment in the window
wnd-size	size of the window ($min(W, rcved\_window)$)
highest-sent	the highest segment transmitted so far
total-sent	Total number of segments transmitted so far
is-idle	does the flow have segments to send?
outstanding-cnt	# of outstanding segments
rtx-timer	when will the rtx timer expire?
user-context	user-defined variables for programmable modules

Table 2: Per-flow state variables in the data delivery engine

from memory to infer segment loss (and perform other user-defined computation as we discuss in §3.3.3).

Periodic Updates. The data delivery engine iterates over the active flows, sending them one at a time to this module to check for retransmission timer expiration and perform other user-defined periodic updates (§3.3.3). Thus, with its 10 ns clock cycle, Tonic can cover each flow within a few microseconds of the expiry of its retransmission timer. This module must be programmable as a retransmission timeout is a signal for detecting loss (#4 in Table 1). Similar to the programmable stage of the Incoming module, the programmers can use per-flow state variables to infer segment loss.

Segment Generation. Given an active flow and its variables, this module generates the next segment's address and forwards it to the credit engine. Tonic can implement segment address generation as a fixed function module based on the following observation (#3 in Table 1): Although different reliable data delivery algorithms have different ways of inferring segment loss, once a lost segment is detected, it is only logical to retransmit it before sending anything new. Thus, the procedure for selecting the next segment is the same irrespective of the data-delivery algorithm, and is implemented as a fixed-function module in Tonic. Thus, this module prioritizes retransmission of lost segments in `marked-for-rtx` over sending the next new segment, i.e., `highest_sent+1` and also increments the number of outstanding segments.

Segment Transmitted. This module is fixed function and is triggered when a segment address is transmitted out of the credit engine. It decrements the number of outstanding segments of the corresponding flow. If the flow was deactivated due to a full ring buffer, it is inserted into the active set again.

3.3 Flexible Credit Management

Transport protocols use congestion-control algorithms to avoid overloading the network by controlling the pace of a flow's transmission. These algorithms consist of a control loop that estimates the network capacity by monitoring the stream of incoming control packets (e.g., acknowledgments and congestion notification packets (CNPs)) and sets parameters that limit outgoing data packets. While the control loop is different in many algorithms, the credit calculation based on parameters is not. Tonic has efficient fixed-function modules for credit calculation (§3.3.1 and §3.3.2) and relegates parameter adjustment to programmable modules (§3.3.3).

3.3.1 Common Credit-Calculation Patterns

Congestion control algorithms have a broad range of ways to estimate network capacity. However, they enforce limits on data transmission in three main ways (#5 in Table 1):

Congestion window. The control loop limits a flow to at most W bytes in flight from the first unacknowledged byte. Thus, if byte i is the first unacknowledged byte, the flow cannot send bytes beyond $i+W$. Keeping track of in-flight segments to enforce a congestion window can get complicated, e.g., in the presence of selective acknowledgments, and is implemented in the fixed-function stage of the incoming module in the data delivery engine.

Rate. The control loop limits the flow's average rate (R) and maximum burst size (D). Thus, if a flow had credit c_0 at the time t_0 of the last transmission, then the credit at time t will be $min(R*(t-t_0)+c_0, D)$. As we show in §4, implementing precise per-flow rate limiters under our strict timing and memory constraints is challenging and has an optimized fixed-function implementation in Tonic.

Grant tokens. Instead of estimating network capacity, the control loop receives tokens from the receiver and adds them to the flow's credit. Thus, the credit of a flow is the total tokens received minus the number of transmitted bytes, and the credit calculation logic consists of a simple addition.

Given that these are used by most congestion control algorithms, we optimize the implementation of each to meet Tonic's timing and memory constraints. Congestion window calculations are mostly affected by acknowledgments, Thus, calculation and enforcement of congestion window happens in the data delivery engine. For the other two credit calculation schemes, Tonic relies on the credit engine to process credit-related event, and Tonic users can simply pick which credit-calculation algorithm to use in the credit engine.

3.3.2 Event Processing for Credit Calculation

Conceptually, three main events can trigger credit calculation for a flow, and the credit engine has different modules to concurrently process them every cycle (Figure 2). First, when a segment address is received from the data delivery engine and is the only one in the flow's ring buffer, the flow could now qualify for transmission or remain idle based on its credit (the Enqueue module). Second, when a flow transmits a segment address, its credit must be decreased and we should determine whether it is qualified for further transmission based on its updated credit and the occupancy of its ring buffer (the Transmit module). Third are events that can add credit to the flow (e.g., from grant tokens and leaky bucket rate limiters), which is where the main difference lies between rate-based and token-based credit calculation.

When using grant tokens, the credit engine needs two dedicated modules to add credit to a flow: one to process incom-

ing grant tokens from the receiver, and one to add credit for retransmissions on timeouts. When using rate, the credit engine does not need any extra modules for adding credit since a flow with rate R bytes-per-cycle implicitly gains R bytes of credit every cycle and, therefore, we can compute in advance when it will be qualified for transmission.

Suppose in cycle T_0, the Transmit module transmits a segment from flow f, and is determining whether the flow is qualified for further transmission. Suppose that f has more segments in the ring buffer but lacks C bytes of credit. The Transmit module can compute when it will have sufficient credit as $T = \frac{C}{R}$ and set up a timer for T cycles. When the timer expires, f definitely has enough credit for at least one segment, so it can be directly inserted into `ready-to-tx`. When f reaches the head of `ready-to-tx` and is processed by the Transmit module again in cycle T_1, the Transmit module can increase f's credit by $(T_1 - T_0) * R - S$, where S is the size of the segment that is transmitted at time T_1 [3]. Note that when using rate, the credit engine must perform division and maintain per-flow timers. We will discuss the hardware implementation of these operations in §4.

3.3.3 Flexible Parameter Adjustment

Congestion control algorithms often have a control loop that continuously monitors the network and adjusts credit calculation parameters, i.e., rate or window size, based on estimated network capacity. Parameter adjustment is either triggered by incoming packets (e.g., acknowledgments and their signals such as ECN or delay in TCP variants and Timely, and congestion notification packets (CNPs) in DCQCN) or periodic timers and counters (timeouts in TCP variants and byte counter and various timers in DCQCN), and in some cases is inspired by segment losses as well (window adjustment after duplicate acknowledgments in TCP).

Corresponding to these triggers, for specifying parameter adjustment logic, Tonic's users can use the programmable stage of the "Incoming" module, which sees all incoming packets, and the "Periodic Updates" module for timers and counters. Both modules are in the data delivery engine and have access to segment status information, in case segment status (e.g., drops) is needed for parameter adjustment. The updated parameters are forwarded to the credit engine.

As we show in §6.1.1, we have implemented several congestion control algorithms in Tonic and their parameter adjustment calculations have finished within our 10 ns clock cycle. Those with integer arithmetic operations did not need any modifications. For those with floating point operations, such as DCQCN, we approximated the operations to a certain decimal point using integer operations. If an algorithm requires high-precision and complicated floating point operations for parameter adjustment that cannot be implemented within one clock cycle [17], the computation can be relegated to a floating-point arithmetic module outside of Tonic. This module can perform the computation asynchronously and store the output in a separate memory, which merges into Tonic through the "Periodic Updates" module.

[3] Similarly, the Enqueue module can set up the timer when it receives the first segment of the queue and the flow lacks credit for its transmission.

3.4 Handling Conflicting Events

Tonic strives to process events concurrently in order to be responsive to events. Thus, if a flow receives more than one event in the same cycle, it allows the event processing modules to process the events and update the flow's state variables, and reconciles the state before writing it back into memory (the Merge modules in Figure 2).

Since acknowledgments and retransmission timeouts are, by definition, mutually exclusive, Tonic discards the timeout if it is received in the same cycle as an acknowledgment for the same flow. This significantly simplifies the merge logic because several variables (window size and retransmission timer period) are *only* modified by these two events and, therefore, will never be concurrently updated. We can resolve concurrent updates for the remaining variables with simple, predefined merge logic. For example, Segment Generation increments the number of outstanding segments, whereas Segment Transmitted decrements it; if both events affect the same flow at the same time, the number does not change. User-defined variables are updated in either the Incoming or the Periodic Updates module, and we rely on the user to specify which updated variables should be prioritized if both updates happen in the same cycle.

3.5 Tonic's Programming Interface

To implement a new transport logic in Tonic, programmers only need to specify (i) which of the three credit management schemes to use, (ii) the loss detection and recovery logic in response to acknowledgments and timeouts, and (iii) congestion-control parameter adjustment in response to incoming packets or periodic timers and counters. The first one is used to pick the right modules for the credit engine, and the last two are inserted into the corresponding programmable stages of the data delivery engine (Figure 2).

To specify the logic for the programmable stage of the Incoming module, programmers need to write a function that receives the incoming packet (acknowledgment or other control signals), the number of newly acknowledged segments, the `acked` bitmap updated with the information in the acknowledgment, the old and new value of `wnd-start` (in case the window moves forward due to a new cumulative acknowledgment), and the rest of the flow's state variables (Table 2) as input. In the output, they can mark a range of segments for retransmission in `marked-for-rtx`, update congestion-control parameters such as window size and rate, and reset the retransmission timer. The programming interface of the Periodic Updates module is similar.

In specifying these functions, programmers can use inte-

ger arithmetic operations, e.g., addition, subtraction, multiplication, and division with small-width operands, conditionals, and a limited set of read-only bitmap operations, e.g., index lookup, and finding the first set bit in the updated `acked` bitmap (see appendix D for an example program). Note that, as we described in §3.2.3, a dedicated fixed-function stage in the data delivery engine performs the costly common bitmap updates on receipt of acknowledgments. We evaluate Tonic's programming interface in §6.1.1, where we show that a wide range of transport protocols can be implemented using this interface and give examples of ones that cannot.

4 Hardware Implementation

In this section, we describe the hardware design of the Tonic components that were the most challenging to implement under Tonic's tight timing and memory constraints.

High-Precision Per-Flow Rate Limiting. A flow with rate R bytes per cycle and C bytes to send will have sufficient credit for transmission in $T = \lceil \frac{C}{R} \rceil$ cycles. Tonic needs to do this computation in the credit engine but must represent R as an integer since it cannot afford to do floating-point division. This creates a trade-off between the rate-limiting precision and the range of rates Tonic can support. If R is in bytes per cycle, we cannot support rates below one byte per cycle or $\sim$1 Gbps. If we represent R in bytes per thousand cycles, we can support rates as low as 1 Mbps. However, $T = \lceil \frac{C}{R} \rceil$ determines how many thousand cycles from now the flow qualifies for transmission which results in lower rate conformance and precision for higher-bandwidth flows. To support a wide range of rates without sacrificing precision, Tonic keeps multiple representations of the flow's rate at different levels of precision and picks the most precise representation for computing T at any moment (details in Appendix B).

Efficient Bitmap Operations. Tonic uses bitmaps as large as 128 bits to track the status of segments for each flow. Bitmaps are implemented as ring buffers. The head pointer corresponds to the first unacked segment and moves forward around the buffer with new acks. To efficiently implement operations whose output depends on the values of *all* the bits in the bitmap, we must divide the buffer into smaller parts in multiple layers, process them in parallel, and join the results. One such operation, frequently used in Tonic, is finding the first set bit after the head. The moving head of the ring buffer complicates the implementation of this operation since keeping track of the head in each layer requires extra processing, making it difficult to compute within our 10 ns target. Instead, Tonic uses a light-weight pre-processing on the input ring buffer to avoid head index computation in the layers altogether (details in Appendix C).

Concurrent Memory Access. The memory in data delivery engine is concurrently accessed by five modules (including both stages of the Incoming module) every cycle (§3.2.3). However, FPGAs only have dual-ported block RAMs, with each port capable of either read or write every cycle. Building memories with more concurrent reads and writes requires keeping multiple copies of data in separate memory "banks" and keeping track of the bank with the most recent data for each address[4] [24]. To avoid supporting *five* concurrent reads and writes, we manage to partition per-flow state variables into two groups, each processed by at most four events. Thus, Tonic can use two memories with four read and write ports instead of a single one with five, to provide concurrent access for all processing modules at the same time.

5 Integrating Tonic into the Transport Layer

This section describes how to integrate Tonic into the Linux Kernel for applications using the Socket API. Tonic's transport logic is intentionally decoupled from the specific implementation of other transport functionality such as connection management, application-level API, and buffer management. While Section 2 provides a high-level overview of Tonic's relationship with the rest of the transport layer, this section provides an illustrative and detailed example of how Tonic can interface with the rest of the transport layer to learn about new connections, requests for data transmission on existing connections, and connection termination. As another example, we discuss how Tonic can be used in RDMA-based transport layers in Appendix A.

After creating and configuring the socket, the application uses multiple system calls for connection management and data transfer. Note that as discussed in §2, Tonic mainly focuses on the sender sider of the transport logic. Thus, only the system calls and modifications relevant to the sender side of the transport layer are discussed in this section.

Connection Management. The system calls include `connect()` on the client to initiate a connection, `listen()` and `accept()` on the server to listen for and accept new connections, and `close()` to terminate a connection. Since connection management happens outside of Tonic, the Kernel implementation of these system calls stays untouched. However, once the connection is established, the Kernel maps it to a unique *flow id* in $[0, N)$, where N is the maximum number of flows supported by Tonic. The Kernel then notifies Tonic through the NIC driver about the new connection. Specifically, from the Transmission Control Block (TCB) allocated for the connection in the Kernel, the IP addresses and ports of the communication endpoints and the maximum segment size (MSS) should be sent to Tonic alongside the flow id. Note that for flows using Tonic for data transfer, the Kernel only needs to track those fields in the TCB that are for connection management (e.g., IP addresses, ports, and TCP FSM), pointers to data buffers, and receiver-related fields. Fields used for data transfer for the sender, i.e., `snd.nxt`, `snd.una`, and `snd.wnd`, are stored in and handled by Tonic.

[4] This overhead is specific to FPGAs, and can potentially be eliminated if the memory is designed as an ASIC.

Finally, after a call to `close()`, the Kernel notifies Tonic of connection termination using the connection's flow id.

Data Transfer. At a high level, `send()` adds more data to the connection's socket buffer, which stores the connection's outstanding data waiting for delivery. As discussed in §3.2, Tonic keeps a few bits of per-segment state for outstanding data and performs all transport logic computation in terms of segments. Therefore, data needs to be partitioned into equal-sized segments before Tonic can start its transmission. As a result, the modifications to the implementation of `send()` mainly involve determining segment boundaries for the data in the socket buffer and deciding when to notify Tonic of the existence of new data segments.

More specifically, the Kernel keeps an extra pointer for each connection's socket buffer, in addition to its `head` and `tail`, called `tonic-tail`. It points to the end of the last data segment of which Tonic has been notified and is used in the segmentation process described below. `head` and updates to `tonic-tail` are sent to Tonic to use when generating the address of the next segment to fetch from memory.

Starting with an empty socket buffer, when the application calls `send()`, data is copied to the socket buffer, and `tail` is updated accordingly. The data is then partitioned into MSS-sized segments. Suppose the data is partitioned into S segments and $B < MSS$ remaining bytes. The Kernel then updates `tonic-tail` to point to the end of the last MSS-sized segment, i.e., `head + MSS * S`, and notifies Tonic of the update to `tonic-tail`. The extra B bytes remain unknown to Tonic for a configurable time T, in case the application calls `send` to provide more data. In that case, the data are added to the socket buffer, data between `tonic-tail` and `tail` are similarly partitioned, `tonic-tail` is updated accordingly, and Tonic is notified of new data segments.

If there is not enough data for a MSS-sized segment after T, the Kernel needs to notify Tonic of the "small" segment and its size, and update `tonic-tail` accordingly. Note that Tonic requires all segments, except for the last one in a burst, to be of equal size, as all computations, including window updates, are in terms of segments. Thus, after creating a "small" segment, if there is more data from the application, Tonic can only start its transmission when it is done transferring its current segments. Tonic notifies the Kernel once it successfully delivers the final "small" segment, at which point, `head` and `tonic-tail` will be equal, and the Kernel continues partitioning the remaining data in the socket buffer and updating Tonic as before. Note that Tonic can periodically forward acknowledgements to the kernel to move `head` forward and free up space for new data in the socket buffer.

Other Considerations. As we show in §6, Tonic's current design supports 2048 concurrent flows, which matches the working sets observed in data centers [14,35] and other hardware offloads in the literature [18]. If a host has more open connections than Tonic can support, the Kernel can offload data transfer for high-bandwidth flows to Tonic on a first-come first-serve basis, or have users set a flag when creating the socket and fall back to software once Tonic runs out of resources for new flows. Alternatively, modern FPGA-based NICs have a large DRAM directly attached to the FPGA [18]. The DRAM can potentially be used to store the state of more connections, and swap them back and forth into Tonic's memory as they activate and need to transmit data. Moreover, to provide visibility into the performance of hardware transport logic, Tonic can provide an interface for Kernel to periodically pull transport statistics from the NIC.

Takeaways. Linux Kernel can be modified so applications can use Tonic through the socket API. That said, Tonic is most beneficial for high-bandwidth flows that generate MSS-sized segments. Flows that sporadically generate small segments do not benefit as much, as small segments cannot be consolidated within Tonic. We emphasize that the above design serves as an example of how Tonic can be integrated into a commonly-used transport layer. However, TCP, sockets, and bytestreams are not always suitable for high-bandwidth, low-latency flows. In fact, several such data-center applications are starting to use RDMA and its message-based API instead [5, 9, 20, 33]. Tonic can be integrated into RDMA-based transport as well, which we discuss in Appendix A.

6 Evaluation

To evaluate Tonic, we implement a prototype in Verilog (∼8K lines of code) and a cycle-accurate hardware simulator in C++ (∼2K lines of code). The simulator is integrated with NS3 network simulator [4] for end-to-end experiments.

To implement a transport protocol on Tonic's Verilog prototype, programmers only need to provide three Verilog files: (i) `incoming.v`, describing the loss detection and recovery logic and how to change credit management parameters (i.e., rate or window) in response to incoming packets; this code is inserted into the second stage of the Incoming pipeline in the data delivery engine, (ii) `periodic_updates.v`, describing the loss detection and recovery logic in response to timeouts and how to change credit management parameters (i.e., rate or window) in response to periodic timers and counters; this code is inserted into the Periodic Updates module in the data delivery engine, and (iii) `user_configs.vh`, specifying which of the three credit calculation schemes to use and the initial values of user-defined state variables and other parameters, such as initial window size, rate, and credit.

We evaluate the following two aspects of Tonic:

Hardware Design (§6.1). We use Tonic's prototype to evaluate its hardware architecture for *programmability* and *scalability*. Can Tonic support a wide range of transport protocols? Does it reduce the development effort of implementing transport protocols in the NIC? Can Tonic support complex user-defined logic with several variables? How many per-flow segments and concurrent flows can it support?

End-to-End Behavior (§6.2). We use Tonic's cycle-accurate

simulator and NS3 to compare Tonic's end-to-end behavior with that of hard-coded implementations of two protocols: New Reno [21] and RoCEv2 with DCQCN [41], both for a single flow and multiple flows sharing a bottleneck link.

6.1 Hardware Design

There are two main metrics for evaluating the efficiency of a hardware design: (i) **Resource Utilization.** FPGAs consist of primitive blocks, which can be configured and connected differently to implement a Verilog program: *look-up tables (LUTs)* are the main reconfigurable logic blocks, *block RAMs (BRAMs)* are used to implement memory. and (ii) **Timing.** At the beginning of each cycle, each module's input is written to a set of input registers. The module must process the input and prepare the result for the output registers before the next cycle begins. Tonic must *meet timing* at 100 MHz to transmit a segment address every 10 ns. That is, to achieve 100 Gbps, the processing delay of every path from input to output registers in every module must stay within 10 ns.

We use these two metrics to evaluate Tonic's programmability and scalability. These metrics are highly dependent on the specific target used for synthesis. We use the Kintex Ultrascale+ XCKU15P FPGA as our target because this FPGA, and others with similar capabilities, are included as bump-in-the-wire entities in today's commercial programmable NICs [2, 3]. This is a conservative choice, as these NICs are designed for 10-40 Gbps Ethernet. A 100 Gbps NIC could potentially have a more powerful FPGA. Moreover, we synthesize *all* of Tonic's components onto the FPGA to evaluate it as a standalone prototype. However, given the well-defined interfaces between the fixed-function and programmable modules, it is conceivable to implement the fixed-function components as an ASIC for more efficiency. Unless stated otherwise, we set the maximum number of concurrent flows to 1024 and the maximum window size to 128 segments in all of our experiments [5].

6.1.1 Hardware Programmability

We have implemented the sender's transport logic of six protocols in Tonic as representatives of various types of segment selection and credit calculation algorithms in the literature. Table 3 summarizes the resource utilization of these Tonic-based implementations for both fixed-function and user-defined modules, as well as the lines of code and bytes of user-defined state it took to implement them.

Reno [12] and New Reno [21] represent TCP variants that use only cumulative acknowledgments for reliable delivery and congestion window for credit management. Reno can only recover from one loss within the window using fast retransmit, whereas New Reno uses partial acknowledgments to recover more efficiently from multiple losses in the same window. SACK, inspired from RFC 6675 [15], represents TCP variants that use selective acknowledgments. Our implementation has one SACK block per acknowledgment but can be extended to more.

[5] A 100 Gbps flow with 1500B back-to-back packets over 15-μs RTT, typical in data centers, has at most 128 in-flight segments.

	User-Defined Logic		Credit Type	Look up Tables (LUTs)				BRAMs	
				User-Defined		Fixed			
	LoC	state(B)		total(K)	%	total(K)	%	total	%
Reno	48	8	wnd	2.4	0.5	109.4	20.9	195	20
NewReno	74	13	wnd	2.6	0.5	112.5	21.5	211	21
SACK	193	19	wnd	3.3	0.6	112.1	21.4	219	22
NDP	20	1	token	3.0	0.6	143.6	29.0	300	30
RoCE w/ DCQCN	63	30	rate	0.9	0.2	185.2	35.2	251	26
IRN	54	14	rate	2.9	0.6	177.4	33.9	219	22

Table 3: Resource utilization of transport protocols in Tonic.

NDP [22] represents receiver-driven protocols, recently proposed for low-latency data-center networks [19,34]. NDP senders use explicit NACKs and timeouts for loss detection and rely on grant tokens for congestion control. RoCEv2 with DCQCN [41] is a widely-used transport for RDMA over Ethernet, and IRN [32] is a recent hardware-based protocol that improves the simple reliable delivery algorithm on RoCE NICs. Both use rate limiters for credit management.

Note that, as described in §3.2, not all data-delivery algorithms are feasible for hardware implementation as is. For instance, due to memory constraints on the NIC, it is not possible to keep timestamps for *every* packet, new and retransmissions, on the NIC. As a result, transport protocols which rely heavily on per-packet timestamps, e.g., QUIC [25], need to be modified to work with fewer timestamps, i.e., for a subset of in-flight segments, to be offloaded to hardware.

Takeways. There are three key takeaways from these results:

- *Tonic supports a variety of transport protocols.*
- *Tonic enables programmers to implement new transport logic with modest development effort.* Using Tonic, each of the above protocols is implemented in less than 200 lines of Verilog code, with the user-defined logic consuming less than 0.6% of the FPGA's LUTs. In contrast, Tonic's fixed-function modules, which are reused across these protocols, are implemented in ∼8K lines of code and consume ∼sixty times more LUTs.
- *Different credit management schemes have different overheads.* For transport protocols that use congestion window for credit management, window calculations overlap with and therefore are implemented in the data delivery engine §3.3.1. As a result, their credit engine utilizes fewer resources (both reconfigurable logic and memory) than others. Rate limiting requires more per-flow state and more complicated operations (§4) than enforcing receiver-generated grant tokens but needs fewer memory ports for concurrent reads and writes (§3.3.2), overall leading to lower BRAM and higher LUT utilization for rate limiting.

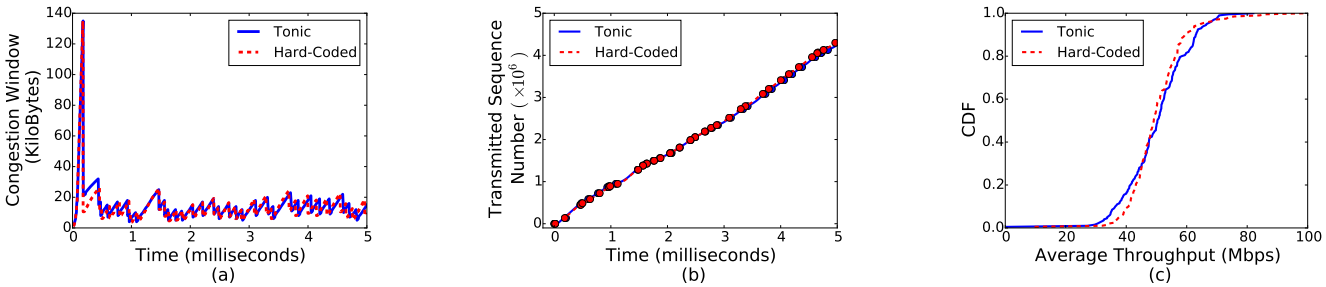

Figure 3: NewReno's Tonic vs hard-coded implementation in NS3 (10G line-rate): a) Congestion window updates (single flow, random drops), b) Transmitted sequence numbers with retransmission in large dots (single flow, random drops), and c) CDF of average throughput of multiple flows sharing a bottleneck link over 5 seconds (200 flows from 2 hosts to one receiver)

6.1.2 Hardware Scalability

To evaluate Tonic's scalability, we examine how sources of variability in its architecture affect memory utilization and timing (Results summarized Table 4).

User-defined logic in programmable modules can have arbitrarily-long chains of dependent operations, potentially causing timing violations. We generate 70 random programs for `incoming.v` (the programmable stage of Incoming module in data delivery engine) with different numbers of arithmetic, logical, and bitmap operations, and analyze how long the chain of dependent operations gets without violating timing at 10ns. These programs use up to 125B of state and have a maximum dependency of 65 *logic levels* (respectively six and two times more than the benchmark protocols in Table 3). Each logic level represents one of several primitive logic blocks (LUT, MUX, DSP, etc.) chained together to implement a path in a Verilog program.

We plug these programs into Tonic, synthesize them, and analyze the relationship between the number of logic levels and latency of the max-delay path in comparison to benchmark programs. As summarized in Table 4, our benchmark protocols have 13 to 29 logic levels on their max-delay path and all meet timing. Synthetic programs with up to 32 logic levels consistently meet timing, while those with more than 43 logic levels do not. Between 32 and 42 logic levels, the latency of the max-delay path is around 10 ns. Depending on the mix of primitives on the max-delay path and their latencies, programs in that region can potentially meet timing. Thus, Tonic not only supports our benchmark protocols, but also has room to support future more sophisticated protocols.

User-defined state variables increase the memory width affecting BRAM utilization. We add extra variables to SACK, IRN, and NDP to see how wide memories can get without violating timing and running out of BRAMs on the FPGA, repeating the experiment for each of the three credit management schemes as they have different memory footprints. As shown in Table 4, programmers can use 448 bytes of user-defined state if they use congestion window, 340 bytes if they use rate, and 256 bytes if they use grant tokens (Benchmark programs in Table 3 use less than 30 bytes).

	Metric	Results	
Complexity of User-Defined Logic	logic levels	$(0, 31]$	meets timing
		$(31, 42]$	depends on operations
		$(42, 65]$	violates timing
User-Defined State	bytes	256	grant token
		340	rate
		448	congestion window
Window Size	segments	256	
Concurrent Flows	count	2048	

Table 4: Summary of Tonic's scalability results.

Maximum window size determines the size of per-flow bitmaps stored in the data delivery engine to keep track of the status of a flow's segments, therefore affecting memory utilization, and the complexity of bitmap operations, hence timing. Tonic can support bitmaps as large as 256 bits (i.e., tracking 256 segments), with which we can support a single 100Gbps flow in a network with up to 30µs RTT.

Maximum number of concurrent flows determines memory depth and the size of FIFOs used for flow scheduling (§3.1). Thus, it affects both memory utilization and the queue operations, hence timing. Tonic can scale to 2048 concurrent flows in hardware which matches the size of the active flow set observed in data centers [14, 35] and other hardware offloads in the literature [18].

Takeways. Tonic has additional room to support future protocols that are more sophisticated with more user-defined variables than our benchmark protocols. It can track 256 segments per flow and support 2048 concurrent flows. With a more powerful FPGA with more BRAMs, Tonic can potentially support even larger windows and more flows.

6.2 End-to-End Behavior

To examine Tonic's end-to-end behavior and verify the fidelity of Tonic-based implementation of transport logic in different transport protocols, we have developed a cycle-accurate hardware simulator for Tonic in C++ and integrated it into NS3. We implement a NewReno and a RoCEv2 with DCQCN sender in our Tonic simulator and demonstrate that the end-to-end behavior of their Tonic-based implementation matches that of their hard-coded implementation in NS3.

Note that our goal in performing these simulations is to an-

alyze and verify Tonic's end-to-end behavior. Tonic's capability to support 100Gbps line rate has been demonstrated in the previous section using hardware synthesis. Thus, in our simulations, we use 10Gbps and 40Gbps as line rate merely to make hardware simulations with multiple flows over seconds computationally tractable.

6.2.1 TCP New Reno

We implement TCP New Reno in Tonic based on RFC 6582, and use NS3's native network stack for the hard-coded implementation of New Reno. Our Tonic-based implementation works with the *unmodified* native TCP receiver in NS3. In all simulations, the hosts are connected via 10Gbps links to the same switch, the RTT is $10\mu s$, the buffer is 5.5MB, the minimum retransmission timeout is 200ms (Linux default), segments are 1000 bytes large, and delayed acknowledgments are enabled on the receiver.

Single Flow. We start a single flow from one host to another, and randomly drop packets on the receiver's NIC. Figure 3.a and 3.b show the updates to the congestion window and transmitted sequence numbers (retransmissions are marked with large dots) respectively. Tonic's behavior in both cases closely matches the hard-coded implementation. The slight differences stem from the fact that in NS3's network stack, all the computation happens in the same virtual time step while in Tonic every event (incoming packets , segment address generation, etc.) is processed over a 100ns cycle (increased from 10ns to match the 10G line rate).

Multiple Flows. Two senders each start 100 flows to a single receiver, so 200 flows share a single bottleneck link for 5 seconds. As shown in Figure 3.c, the CDF of average throughput across the 200 flows for the Tonic-based implementation closely matches that of the hard-coded implementation. We observe similarly matching distributions for number of retransmissions. When analyzing the flows' throughput in millisecond-long epochs, we notice larger variations in the hard-coded implementation than Tonic since Tonic, as opposed to NS3's stack, performs per-packet round robin scheduling across flows on the same host.

6.2.2 RoCEv2 with DCQCN

We implement RoCE with DCQCN based on [41], and use the authors' NS3 implementation from [42] for the hard-coded implementation. Our Tonic-based implementation works with the *unmodified* hard-coded RoCE receiver. In all simulations, hosts are connected via 40Gbps links to the same switch, RTT is $4\mu s$, segments are 1000B large, and we use the default DCQCN parameters from [42].

Single Flow. DCQCN is a rate-based algorithm which performs congestion control using CNPs and periodic timers and counters as opposed to packet loss in TCP. Thus, to observe rate updates for a single flow, we run two flows from two different hosts to the same receiver for one second to create congestion and track the throughput changes of one as they both converge to the same rate. As shown in Figure 4, Tonic's behavior in terms of rate updates closely matches the hard-coded implementation. Moreover, we ran a single DCQCN flow at 100Gbps with 128B back-to-back packets and confirmed that Tonic can saturate the 100Gbps link.

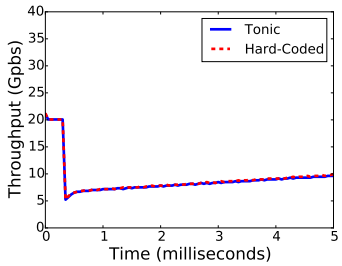

Figure 4: RoCEv2 with DCQCN in Tonic vs hard-coded in NS3 (40G line rate, one of two flows on a bottleneck link).

Multiple Flows. Two senders each start 100 flows to a single receiver, so 200 flows share a single bottleneck link for one second. Both Tonic and the hard-coded implementation do per-packet round robin scheduling among the flows on the same host. As a result, all flows in both cases end up with an average throughput of $203 \pm 0.2 Mbps$. Moreover, we observe a matching distribution of CNPs in both cases.

7 Related Work

Tonic is the first programmable architecture for transport logic in hardware able to support 100 Gbps. In this section, we review the most closely related prior work.

Commercial hardware network stacks. Some newer NICs have hardware network stacks, including hard-wired transport protocols [8, 10]. However, these NICs only implement two transport protocols, either RoCE [8] or a vendor-selected variant of TCP, and they cannot be modified without going through the vendor. Tonic enables programmers to implement a variety of transport protocols in hardware with modest effort. Since a detailed description of the architecture of these commercial NICs is not publicly available, we were not able to compare our design decisions with theirs.

Non-commercial hardware transport protocols. Recent work explores hardware transport protocols that run at high speed with low memory footprint [28, 29, 32]. Tonic facilitates innovation in this area by enabling researchers to implement new protocols with modest development effort.

Accelerating network functionality. Several academic and industrial projects offload end-host virtual switching and network functions to FPGAs, processing a stream of already-generated packets [13, 18, 26, 27, 39]. Tonic, on the other hand, implements the transport logic in the NIC by keeping track of potentially a few hundred segments at a time to generate packets at line rate while running user-defined transport logic to ensure efficient and reliable delivery.

References

[1] F-Stack. http://www.f-stack.org/. Accessed: February 2019.

[2] Innova Flex 4 Lx EN Adapter Card. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova_Flex4_Lx_EN.pdf. Accessed: February 2019.

[3] Mellanox Innova 2 Flex Open Programmable SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. Accessed: February 2019.

[4] NS3 Network Simulator. https://www.nsnam.org/. Accessed: February 2019.

[5] NVMe over Fabric. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf. Accessed: February 2019.

[6] OpenOnload. https://www.openonload.org/. Accessed: February 2019.

[7] RDMA - iWARP. https://www.chelsio.com/nic/rdma-iwarp/. Accessed: February 2019.

[8] RDMA and RoCE for Ethernet Network Efficiency Performance. http://www.mellanox.com/page/products_dyn?product_family=79&mtag=roce. Accessed: February 2019.

[9] RoCE Accelerates Data Center Performance, Cost Efficiency, and Scalability. http://www.roceinitiative.org/wp-content/uploads/2017/01/RoCE-Accelerates-DC-performance_Final.pdf. Accessed: February 2019.

[10] TCP Offload Engine (TOE). https://www.chelsio.com/nic/tcp-offload-engine/. Accessed: February 2019.

[11] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[12] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control . RFC 5681.

[13] ARASHLOO, M. T., GHOBADI, M., REXFORD, J., AND WALKER, D. Hotcocoa: Hardware congestion control abstractions. In *HotNets* (2017).

[14] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[15] BLANTON, E., ALLMAN, M., WANG, L., JARVINEN, I., KOJO, M., AND NISHIDA, Y. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. RFC 6675.

[16] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-Based Congestion Control. *ACM Queue* (2016).

[17] DONG, M., LI, Q., ZARCHY, D., GODFREY, P. B., AND SCHAPIRA, M. PCC: Re-architecting Congestion Control for Consistent High Performance. NSDI'15.

[18] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI* (2018).

[19] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric. In *CoNEXT* (2015).

[20] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over Commodity Ethernet at Scale. In *SIGCOMM* (2016).

[21] HANDERSON, T., FLOYD, S., GURTOV, A., AND NISHIDA, Y. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582.

[22] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WÓJCIK, M. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM* (2017).

[23] JEONG, E., WOO, S., JAMSHED, M. A., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI* (2014).

[24] LAFOREST, C. E., AND STEFFAN, J. G. Efficient Multi-Ported Memories for FPGAs. In *FPGA* (2010).

[25] LANGLEY, A., RIDDOCH, A., WILK, A., VICENTE, A., KRASIC, C., ZHANG, D., YANG, F., KOURANOV, F., SWETT, I., IYENGAR, J., ET AL. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM* (2017).

[26] LAVASANI, M., DENNISON, L., AND CHIOU, D. Compiling High Throughput Network Processors. In *FPGA* (2012).

[27] LI, B., TAN, K., LUO, L. L., PENG, Y., LUO, R., XU, N., XIONG, Y., CHENG, P., AND CHEN, E. Clicknp: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM* (2016).

[28] LU, Y., CHEN, G., LI, B., TAN, K., XIONG, Y., CHENG, P., ZHANG, J., CHEN, E., AND MOSCIBRODA, T. Multi-Path Transport for {RDMA} in Datacenters. In *NSDI* (2018).

[29] LU, Y., CHEN, G., RUAN, Z., XIAO, W., LI, B., ZHANG, J., XIONG, Y., CHENG, P., AND CHEN, E. Memory Efficient Loss Recovery for Hardware-Based Transport in Datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking* (2017).

[30] MARINOS, I., WATSON, R. N., AND HANDLEY, M. Network Stack Specialization for Performance. In *SIGCOMM* (2014).

[31] MATHIS, M., AND MAHDAVI, J. Forward Acknowledgement: Refining TCP Congestion Control. In *SIGCOMM* (1996).

[32] MITTAL, R., SHPINER, A., PANDA, A., ZAHAVI, E., KRISHNAMURTHY, A., RATNASAMY, S., AND SHENKER, S. Revisiting Network Support for RDMA. In *SIGCOMM* (2018).

[33] MITTAL, R., THE LAM, V., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-Based Congestion Control for the Datacenter. In *SIGCOMM* (2015).

[34] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM* (2018).

[35] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *SIGCOMM* (2015).

[36] SAEED, A., DUKKIPATI, N., VALANCIUS, V., CONTAVALLI, C., AND VAHDAT, A. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM* (2017).

[37] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queuing Using Deficit Round-Robin. *Transactions on Networking* (1996).

[38] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *SIGCOMM* (2016).

[39] SULTANA, N., GALEA, S., GREAVES, D., WÓJCIK, M., SHIPTON, J., CLEGG, R., MAI, L., BRESSANA, P., SOULÉ, R., MORTIER, R., ET AL. Emu: Rapid Prototyping of Networking Services. In *ATC* (2017).

[40] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM* (2011).

[41] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM* (2015).

[42] ZHU, Y., GHOBADI, M., MISRA, V., AND PADHYE, J. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *CoNEXT* (2016).

A Integrating Tonic within RDMA

Remote Direct Memory Access (RDMA) enables applications to directly access memory on remote endpoints without involving the CPU. To do so, the endpoints create a *queue pair*, analogous to a connection, and post requests, called *Work Queue Elements (WQEs)*, for sending or receiving data from each other's memory. Although RDMA originated from InfiniBand networks, RDMA over Ethernet is getting more common in data centers [9, 20, 33]. In the rest of this section, we use RDMA to refer to RDMA implementations over Ethernet.

Once a queue pair is created, RDMA NICs can add the new "connection" to Tonic and use it to *on the sender side* to transfer data in response to different WQEs. Each WQE corresponds to a separate message transfer and therefore nicely fits Tonic's need for partitioning data into segments before starting transmission.

For instance, in an RDMA Write, one endpoint posts a Request WQE to write to memory on the other endpoint. Data length, data source address on the sender, and data sink addresses on the receiver are specified in the Request WQE. Thus, a shim layer between RDMA applications and Tonic can break the data into segments and notify Tonic of number of segments, and the source memory address to read the data from on the sender. Once Tonic generates the next segment address, the rest of the RDMA NIC should DMA it from the sender's memory and add appropriate headers.

An RDMA Send is similar to RDMA Write, except it requires a Receive WQE on the receiver to specify the sink address to which the data from the sender should be written. So, the sender side can still use Tonic in the same way. As another example, in an RDMA Read, one endpoint requests data from memory on the other endpoint. So the responder endpoint should transmit data to the requester endpoint. Again, the data length, data source on the responder, and data sink on the requester are specified in the WQE, and the shim layer can break it into segments and transfer it using Tonic.

Thus, Tonic can be integrated into RDMA NICs to replace the hard-coded transport logic on the sender-side of data transfer. In fact, two of our benchmark protocols, RoCE with DCQCN [41] and IRN [32] are proposed for RDMA NICs. That said, this is assuming we have a compatible receiver on the other receiver-side to generate the control signals (e.g., acknowledgements, congestion notifications, etc.) required by whichever transport protocol one chooses to implement on Tonic on the sender side.

While some implementations of RDMA over Ethernet such as iWarp [7] handle out-of-order (OOO) packets and implement TCP/IP-like acknowledgments, others namely RoCE [8] assume a lossless network and have simpler transport protocols that do not require receivers to handle OOO packets and generate frequent control signals. However, as RDMA over Ethernet is getting more common in data centers, the capability to handle OOO packets on the receiver and generate various control signals for more efficient transport is being implemented in these NICs as well [32, 41].

Takeaways. Tonic can be integrated into RDMA NICs to replace the hard-coded transport logic on the sender-side of data transfer.

B High-Precision Per-Flow Rate Limiting

When using rate in the credit engine, if a flow with rate R bytes per cycle needs C more bytes of credit to transmit a segment, Tonic calculates $T = \lceil \frac{C}{R} \rceil$ as the time where the flow will have sufficient credit for transmission. It sets up a timer that expires in T cycles, and upon its expiry, queues up the flow in `ready-to-tx` for transmission (§3.3.2). Note that T must be calculated in the fast path. Since we cannot afford to do floating-point division in the fast path, R must be represented as an integer.

This creates a trade-off between the rate-limiting precision and the range of rates Tonic can support. If we represent R in bytes per cycle, we can compute the exact cycle when the flow will have enough credit, but cannot support rates lower than one byte per cycle or $\sim$1 Gbps. If we instead represent R in, say, bytes per thousand cycles, we can support lower rates (e.g., 1 Mbps), but $T = \lceil \frac{C}{R} \rceil$ will determine how many thousand cycles from now the flow can qualify for transmission. This results in lower rate conformance and precision for higher-bandwidth flows. As a concrete example, for a 20 Gbps flow, R would be 25000 bytes per thousand cycles. Suppose the flow has a 1500-byte segment to transmit. It will have enough credit to do so in 8 cycles but has to wait $\lceil \frac{1500}{25000} \rceil = 1$ thousand cycles to be queued for transmission.

Instead of committing to one representation for R, Tonic keeps multiple variables $R_1, \ldots, R_k$ for each flow, each representing flow's rate at a different level of precision. As the congestion control loop adjusts the rate according to network capacity, Tonic can efficiently switch between $R_1, \ldots, R_k$ to pick the most precise representation for computing T at any moment. This enables Tonic to support a wide range of rates without sacrificing the rate-limiting precision.

C Efficient Bitmap Operations

Tonic uses bitmaps as large as 128 bits to track the status of a window of segments for each flow. Bitmaps are implemented as ring buffers, with the head pointer corresponding to the first unacknowledged segment. As new acknowledgments arrive, the head pointer moves forward around the

ring. To efficiently implement operations whose output depends on the values of *all* the bits in the bitmap, we must parallelize them by dividing the ring buffer into smaller parts, processing them in parallel, and joining the results. For large ring buffers, this divide and conquer pattern is repeated in multiple layers. As each layer depends on the previous one for its input, we must keep the computation in each layer minimal to stay within our 10 ns target.

One such operation finds the first set bit after the head. This operation is used to find the next lost segment for retransmission in the `marked-for-rtx` bitmap. The moving head of the ring buffer complicates the implementation of this operation. Suppose we have a 32-bit ring buffer A_{32}, with bits 5 and 30 set to one, and the head at index 6. Thus, $findfirst(A_{32}, 6) = 30$. We divide the ring into eight four-bit parts, "or" the bits in each one, and feed the results into an 8-bit ring buffer A_8, where $A_8[i] = OR(A_{32}[i : i+3])$. So, only $A_8[1]$ and $A_8[7]$ are set. However, because the set bit in $A_{32}[4 : 7]$ is *before* the head in the original ring buffer, we cannot simply use one as A_8's head index or we will mistakenly generate 5 instead of 30 as the final result. So, we need extra computation to find the correct new head. For a larger ring buffer with multiple layers of this divide and conquer pattern, we need to compute the head in each layer.

Instead, we use a lightweight pre-processing on the input ring buffer to avoid head index computation altogether. More specifically, using A_{32} as input, we compute A'_{32} which is equal to A_{32} except that all the bits from index zero to head (6 in our example) are set to zero. Starting from index zero, the first set bit in A'_{32} is always closer to the original head than the first set bit in A_{32}. So, $findfirst(A_{32}, 6)$ equals $findfirst(A'_{32}, 0)$ if A'_{32} has any set bits, and otherwise $findfirst(A_{32}, 0)$. This way, independent of the input head index H, we can always solve $findfirst(A, H)$ from two subproblems with the head index *fixed* at zero.

D New Reno in Tonic

The following is the implementation of New Reno's loss detection and recovery algorithm on receipt of acknowledgments in Tonic [21]. Extra comments have been added for clarification.

```
module new_reno_incoming(
  /************************ INPUTS *****************************/
  // ACK, NACK, SACK, CNP, etc...
  input   [`PKT_TYPE_W-1:0]       pkt_type,
  input   [`PKT_DATA_W-1:0]       pkt_data_in,

  // Segment ID in the cumulative acknowledgment
  input   [`SEGMENT_ID_W-1:0]     cumulative_ack,

  // Segment ID that is selectively acknowledged, if any
  input   [`SEGMENT_ID_W-1:0]     selective_ack,

  // Number of segments acknowledged with the received acknowledgment
  input   [`WINDOW_INDEX_W-1:0]   newly_acked_cnt,

  // Segment ID at the beginning of the window, before and after the
  // acknowledgment
  input   [`WINDOW_INDEX_W-1:0]   old_wnd_start,
  input   [`WINDOW_INDEX_W-1:0]   new_wnd_start,

  // Current time in nanoseconds
  input   [`TIME_W-1:0]           now,

  //// Per-Flow State

  input   [`MAX_WINDOW_SIZE-1:0]  acked,
  input   [`MAX_TX_CNT_SIZE-1:0]  tx_cnt,
  input   [`SEGMENT_ID_W-1:0]     highest_sent,
  input   [`SEGMENT_ID_W-1:0]     wnd_start,
  input   [`WINDOW_SIZE_W-1:0]    wnd_size_in,
  input   [`TIEMR_W-1:0]          rtx_timer_amount_in,
  input   [`SEGMENT_ID_W-1:0]     total_tx_cnt,

  input   [`USER_VARS_W-1:0]      user_vars_in,

  /************************ OUTPUTS ****************************/
  output  [`FLAG_W-1:0]           mark_any_for_rtx,
  output  [`SEGMENT_ID_W-1:0]     mark_for_rtx_from,
  output  [`SEGMENT_ID_W-1:0]     mark_for_rtx_to,
  output  [`WINDOW_SIZE_W-1:0]    wnd_size_out,
  output  [`TIMER_W-1:0]          rtx_timer_amount_out,
  output  [`FLAG_W-1:0]           reset_rtx_timer,

  output  [`USER_VARS_W-1:0]      user_vars_out
);

/************************ Local Variables ********************
*
* Declarations ommited for brevity
*
*************************************************************/

/// is the ack new or duplicate?
assign is_dup_ack   = old_wnd_start == cumulative_ack;
assign is_new_ack   = new_wnd_start > old_wnd_start;

/// count duplicated acks
assign dup_acks     = is_new_ack ? 0:
                      is_dup_ack ? dup_acks_in + 1 : dup_acks_in;

// How many in_flight packets?
assign sent_out     = highest_sent - wnd_start;
assign in_flight    = sent_out - dup_acks;

// update previous highest ack
assign  prev_highest_ack_out = is_new_ack ? old_wnd_start : prev_highest_ack_in;

/// Should we do fast rtx?
assign do_fast_rtx = dup_acks == `DUP_ACKS_THRESH &
                     ((cumulative_ack > recover_in) |
                      (wnd_size_in > 1 & cumulative_ack - prev_highest_ack_in <= 4));

// if yes, update recovery sequence and updated ssh_thresh
assign recovery_seq_out = do_fast_rtx ? highest_sent : recovery_seq_in;

```

```
76  assign half_wnd       = in_flight > 2 ? in_flight >> 1 : 2;
77  assign ss_thresh_out  = do_fast_rtx ? half_wnd : ss_thresh_in;
78
79  //// if in recovery and this is a new ack, is it a
80    // full ack or a partial ack? (Definition in RFC)
81  assign full_ack = is_new_ack & cumulative_ack > recover_in;
82  assign partial_ack = is_new_ack & cumulative_ack <= recover_in;
83
84  // mark for retransmission
85  assign mark_any_for_rtx = do_fast_rtx | partial_ack;
86
87  assign rtx_start = wnd_start_in;
88  assign rtx_end = wnd_start_in + 1;
89
90  // reset rtx timer if not in recovery
91  assign in_recovery_out = do_fast_rtx | (in_recovery_in & cumulative_ack <= recover_in);
92  assign reset_rtx_timer                = ~in_recovery_out;
93
94
95  assign in_timeout_out                 = (~full_ack) & in_timeout_in;
96
97  //// decide new window size
98
99  // keep a counter for additive increase
100 assign additive_inc_cntr_out          = in_recovery_out & ~in_timeout_in ? 0 :
101                                         is_new_ack & wnd_size_in >= ss_thresh_in ?
102                                               (additive_inc_cntr_in == wnd_size_in ? 0 :
103                                                additive_inc_cntr_in + 1): additive_inc_cntr_in;
104
105
106 assign wnd_size_out = new_wnd_size >= `MAX_WINDOW_SIZE ? `MAX_WINDOW_SIZE : new_wnd_size;
107
108 always @(*) begin
109   if (do_fast_rtx) begin
110     // set it equals to new ss_thresh, expanded for performance reasons
111     cwnd_out = sent_out - `DUP_ACKS_THRESH > 2 ? sent_out >> 1 : 1;
112   end
113   else if (~in_recovery_in & is_new_ack) begin
114     if (cwnd_in < ss_thresh_out) begin
115       cwnd_out = cwnd_in + newly_acked_cnt;
116     end
117     else if (wnd_inc_cntr_in >= cwnd_in) begin
118       cwnd_out = cwnd_in + 1;
119     end
120     else begin
121       cwnd_out = cwnd_in;
122     end
123   end
124   else begin
125     cwnd_out = cwnd_in;
126   end
127 end
128 assign there_is_more = in_flight >= cwnd_in;
129
130 always @(*) begin
131   if (do_fast_rtx) begin
132     new_wnd_size = sent_out;
133   end
134   else if (~in_recovery_in & is_new_ack) begin
135     new_wnd_size = cwnd_out;
136   end
137   else begin
138     new_wnd_size = there_is_more ? sent_out : cwnd_in + dup_acks;
139   end
140 end
141
142 //// break up user context into variables
143 assign {prev_highest_ack_in, in_recovery_in, recover_in,
144         in_timeout_in, wnd_inc_cntr_in, ss_thresh_in,
145         dup_acks_in, cwnd_in} = user_cntxt_in;
146
147 assign user_cntxt_out =  {prev_highest_ack_out, in_recovery_out, recover_out,
148                           in_timeout_out, wnd_inc_cntr_out, ss_thresh_out,
149                           dup_acks_outm, cwnd_out};
150
151
152 endmodule
```