SIRD: A Sender-Informed, Receiver-Driven Datacenter Transport Protocol

Konstantinos Prasopoulos, *EPFL;* Ryan Kosta, *UCSD;* Edouard Bugnion, *EPFL;*
Marios Kogias, *Imperial College London*

https://www.usenix.org/conference/nsdi25/presentation/prasopoulos

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

SIRD: A Sender-Informed, Receiver-Driven Datacenter Transport Protocol

Konstantinos Prasopoulos
EPFL

Ryan Kosta
UCSD *

Edouard Bugnion
EPFL

Marios Kogias
Imperial College London

Abstract

Datacenter congestion control protocols are challenged to navigate the throughput-buffering trade-off while relative packet buffer capacity is trending lower year-over-year. In this context, receiver-driven protocols — which schedule packet transmissions instead of reacting to congestion — excel when the bottleneck lies at the ToR-to-receiver link. However, when multiple receivers must use a shared link (*e.g.,* ToR to Spine), their independent schedules can conflict.

We present SIRD, a receiver-driven congestion control protocol designed around the simple insight that single-owner links should be scheduled, while shared links should be managed with reactive control algorithms. The approach allows receivers to both precisely schedule their downlinks *and* to coordinate over shared bottlenecks. Critically, SIRD also treats sender uplinks as shared links, enabling the flow of congestion feedback from senders to receivers, which then adapt their scheduling to each sender's real-time capacity. This results in tight scheduling, enabling high bandwidth utilization with little contention, and thus minimal latency-inducing buffering in the fabric.

We implement SIRD on top of the Caladan stack and show that SIRD's asymmetric design can deliver 100Gbps in software while keeping network queuing minimal. We further compare SIRD to state-of-the-art receiver-driven protocols (Homa, dcPIM, and ExpressPass) and production-grade reactive protocols (Swift and DCTCP) and show that SIRD is uniquely able to simultaneously maximize link utilization, minimize queuing, and obtain near-optimal latency.

1 Introduction

Datacenter workloads like ML training [32, 62] and disaggregated resource management [30] increasingly demand high-throughput and low-latency networking. While datacenter hardware continues to offer higher link speeds, switch packet buffer capacity is failing to keep up, and SRAM density trends show that this is unlikely to change [17, 24]. The combination imposes a challenge for congestion control, which generally relies on buffering to deliver high throughput. The challenge is compounded by other important concerns such as hardware heterogeneity and cost of operations [23, 35].

The most established approach to managing congestion is by reactively slowing down sender transmission rates after detecting a problem. This is the approach of sender-driven (SD) protocols [4, 8, 9, 12, 44, 47, 50, 54, 67, 69, 73, 74] like DCTCP [8] and Swift [44] in which senders make decisions based on network feedback. The reactive nature of SD protocols often requires several roundtrips to address congestion; a limitation for workloads dominated by small flows [16, 33, 56]. Further, the flow abstraction makes it difficult to reduce latency through message-centric scheduling [10, 13].

To address these limitations, a recent line of work [16, 20, 31, 36, 39, 56, 61] proactively schedules packet transmissions instead of reacting to congestion buildup. Scheduling is typically implemented by having receivers send *credit* packets to senders, which the latter consume to send back data. This enables tight control over ToR-to-Host ports which are usually the most congested [33, 65, 72]. The receiver-driven (RD) approach enables high throughput with limited buffering in dcPIM [16] and near-optimal latency in Homa [56].

The fundamental tension in the design of receiver-driven schemes is how to schedule packets over shared links. Each receiver has exclusive control of its downlink but must share the bandwidth of the network's core and that of sender uplinks with other receivers. In a distributed protocol, this creates scheduling conflicts if receivers do not coordinate. Proposed solutions for this problem include explicit pre-matching of senders and receivers (dcPIM [16]), in-network credit throttling (ExpressPass [20]), and the overcommitment of receiver downlinks (Homa [56]). Despite impressive results, each of these approaches makes a sacrifice in either message latency, protocol complexity, or packet buffer utilization.

We propose SIRD, a receiver-driven design based on the following simple insight: exclusive links (receiver downlinks) should be managed proactively, and shared links (sender up-

*Work done while at EPFL.

links and switch-to-switch links) should be managed reactively. In practice, this means that receivers gather congestion feedback from shared links which gives them the means to coordinate, while still explicitly controlling their downlinks. By deducing the bandwidth availability of shared links, and especially that of senders, receivers can be more precise in their credit allocation. This allows them to drive high link utilization without inducing contention and the corresponding queuing in the network. Low queuing, in turn, enables fast message delivery as the network does not introduce head-of-line blocking.

SIRD implements receiver-driven scheduling that reacts to shared-link congestion feedback using 1) end-host signals to detect congestion on sender uplinks, and 2) ECN [63] to detect congestion in the network's core. Congestion information flows via data packets to receivers and is fed into two independent control loops, with the most congested determining how much credit can be allocated to any given sender. SIRD's design is end-to-end and does not require the use of in-network priority queues or unconventional switch configuration.

We implement SIRD in 4300 SLOC within Caladan [29], a state-of-the-art dataplane operating system. We show that SIRD can efficiently allocate credit, minimize queuing, and implement round-robin and SRPT policies at 100Gbps.

We use network simulation to systematically compare SIRD to three state-of-the-art proactive protocols (Homa [56], dcPIM [16], and ExpressPass [20]) and two production-grade reactive protocols (DCTCP [8] and Swift [44]). Our evaluation shows that SIRD can simultaneously maximize link utilization, minimize queuing, and obtain near-optimal latency. Specifically: (1) SIRD causes 12× less peak top-of-rack (ToR) buffering than Homa yet achieves competitive latency and utilization. (2) Compared to dcPIM, SIRD has no message exchange rounds before sending and outperforms it in link utilization, peak ToR buffering, and tail latency by 9%, 43%, and 46%. (3) SIRD achieves 10× lower tail latency and 26% higher utilization than ExpressPass. (4) SIRD outperforms DCTCP and Swift across the board, especially in incast-heavy scenarios.

2 Background

The main goal of congestion control (CC) is to allocate network resources in a work-conserving manner while avoiding packet loss. There are also other, at times conflicting, CC goals, *e.g.,* fairness, flow completion time or deadlines, and multi-tenancy.

Existing datacenter CC schemes for Ethernet networks can be broadly split into two categories: reactive and proactive. In reactive (also sender-driven (SD)) schemes [8–10, 44, 47, 50, 54, 67, 74], senders control congestion by participating in a distributed coordination process to determine the appropriate transmission rates. Senders obtain congestion information through *congestion signals* like ECN [8, 63, 67], delay [44, 54], and in-network telemetry [47, 69].

In proactive schemes [5, 13, 16, 20, 31, 36, 46, 56, 61] bandwidth is allocated explicitly, either globally, or in a distributed manner. Global approaches [46, 61], use a centralized arbiter that controls all transmissions and hence face scalability challenges. Distributed proactive designs operate either end-to-end [13, 16, 31, 56] or with switch involvement in credit management [5, 20]. In end-to-end protocols, which are receiver-driven (RD), each receiver explicitly schedules its downlink by transmitting special-purpose credit tokens to senders. The credit rate can be explicitly controlled by a pacer [20, 36] or be self-clocked [16, 31, 56].

2.1 Exclusive and shared links

Packets flowing to a receiver host can experience congestion either at the ToR-host link, which is exclusive to the receiver, or at host-ToR uplinks and the network core, which are shared among multiple receiving hosts.

Exclusive links: In the context of RD protocols, ToR-to-Host links (downlinks) can be seen as exclusively controlled by a single entity, the receiver. By controlling the rate of credit transmission, a receiver can explicitly control the rate of data arrival - assuming the bottleneck is the downlink. In fact, downlinks are the most common point of congestion in datacenter networks [33, 44, 65, 72]. Congestion at the ToR downlink is the result of incast traffic from multiple senders to one receiver. In turn, incast is the result of the fan-out/fan-in patterns of datacenter applications [25].

RD schemes excel in managing incast traffic because each receiver explicitly controls the arrival rate of data [13, 16, 20, 31, 36, 56]. This level of control also allows RD schemes to precisely dictate which message should be prioritized at downlinks and can even factor-in application requirements [60]. Recent work has delivered significant message latency improvements by scheduling messages based on their remaining size (SRPT policy) [31, 39, 56]. In contrast, SD schemes [8–10, 44, 47, 50, 54] treat downlinks as any other link. Senders must first detect incast at a receiver's downlink and then independently adjust their rates/windows such that the level of congestion falls below an acceptable target.

Shared links: Unlike exclusive downlinks, core and sender-ToR link bottlenecks pose a challenge for RD schemes. Shared link bottlenecks can appear when multiple receivers concurrently pull data packets over the same link. Before discussing existing approaches to deal with this key challenge, we delve deeper into the specifics of shared-link congestion.

When congestion occurs in the network *core*, multiple flows from unrelated senders and receivers compete for bandwidth on core links. Note that although the core may comprise multiple tiers, it is fundamentally shared infrastructure. Congestion at the core of a fabric is less common than at down-

links [33, 56, 65] but can still occur due to core network oversubscription. Oversubscription can be permanent to reduce cost [65] or transient due to component failures. Congestion at the core can also occur because of static ECMP IP routing decisions that cause multiple flows to saturate one core switch and one core-Tor downlink, while other core switches have idle capacity [7, 35].

Uplink congestion occurs due to the fan-out of multiple flows to different receivers or due to bandwidth mismatch between the sender's uplink and the receiver's downlink. Because the resulting packet buffering is in hosts and not in the fabric, sender congestion is a less severe problem from the perspective of packet loss. For RD schemes, uplink congestion is known as the *unresponsive sender problem* [56] and leads to degraded throughput as independent scheduling decisions of receivers may conflict, wasting downlink bandwidth. For example, two receivers may both decide to schedule the same sender A at the same time, even if one of the receivers could be receiving data from sender B. Whereas the term *unresponsive* has been used in the context of SRPT scheduling, where messages are transmitted to completion, we will use *congested sender* as a general description of uplink congestion.

RD protocols have proposed various mechanisms for overcoming the tension between independent receiver scheduling and the fact that some links are shared. One of the earliest designs, pHost [31], employs a timeout mechanism at receivers to detect unresponsive senders and direct credit to other senders. NDP [36] employs very shallow buffer switches and eagerly drops packets, using packet trimming to recover quickly. NDP receivers handle uplink sharing by only crediting senders if they transmit data packets - whether dropped or not. pHost and NDP do not explicitly deal with core congestion and, further, analysis by Montazeri *et al.* [56] showed that neither of the two achieves high overall link utilization. Homa [56] introduced *controlled overcommitment* in which each receiver can send credit to up to k senders at a time. Homa achieves high utilization as it is statistically likely that at least one of the k senders will respond. However, it trades queuing for throughput by meaningfully increasing the amount of expected inbound traffic to each receiver. To let short messages bypass network queues, Homa leverages switch priority queues, which are typically used for application-level QoS guarantees [12, 37, 44]. dcPIM [16] employs a semi-synchronous round-based matching algorithm where senders and receivers exchange messages to achieve a bipartite matching. This coordinates the sharing of sender uplinks but congestion at the core is only implicitly addressed by the protocol's overall low link contention. The downside of this link-sharing approach is message latency. dcPIM delivers small messages quickly by excluding them from the matching process. However, messages larger than the bandwidth-delay-product (BDP) of the network must wait for several RTTs before starting transmission. ExpressPass [20] manages all links, exclusive and shared, via a hop-by-hop approach which configures switches to drop excess credit packets, which in turn rate limits data packets in the opposite direction. To reduce credit drops, ExpressPass uses recent credit drop rates as feedback to adjust the future sending rate, and in this way also improves utilization and fairness across multiple bottlenecks. Out of the designs discussed so far, ExpressPass is the only one that explicitly manages core congestion and can operate with highly oversubscribed topologies. ExpressPass's hop-by-hop design helps it achieve near-zero queuing [20] but is more complex to deploy and maintain due to its switch configuration and path symmetry requirements.

2.2 The impact of ASIC trends on buffering

Congestion control protocols depend on buffering to offset coordination and control loop delays and, as a result, face a throughput-buffering trade-off. Maximum bandwidth utilization can trivially be achieved with high levels of in-network buffering, but at the cost of queuing-induced latency and expensive dropped packet retransmissions. Conversely, low buffering can lead to throughput loss for protocols that are slow in capturing newly available bandwidth.

High-speed packet buffering is handled by small SRAM buffers in switch ASICs, the size of which is not increasing as fast as bisection bandwidth. For example, nVidia's top-end Spectrum 4 ASIC has a 160MB buffer, which corresponds to 3.13MB per Tbps of bisection bandwidth [58]. The previous 12.8Tbps and 6.4Tbps top-end Spectrum ASICs were equipped with 5MB and 6.6MB per Tbps respectively [52, 53] (full list in Appendix A). Unfortunately, future scaling of SRAM densities appears unlikely given CMOS process limitations [17, 24]. In parallel, datacenter round-trip times (RTT) are not falling as they are dominated by host software processing, PCIe latency, and ASIC serialization latencies. Consequently, as link speeds increase, CC protocols must handle higher BDPs with less switch buffer space at their disposal.

To better absorb instantaneous bursts of traffic, switch packet buffers are generally shared among egress ports. Some ASICs advertise fully shareable buffers while others implement separate pools or statically apportion some of the space to each port [33]. On top of the physical implementation, various buffer-sharing algorithms dynamically limit each port's maximum allocation to avoid unfairness [8]. However, if a large part of the overall buffer is occupied, the per-port cap becomes more equitable, limiting burst absorbability [33].

Figure 1 provides some context by comparing the buffer use of Homa, a state-of-the art RD protocol, with recent switch buffer capacities under the Websearch workload [10]. The dotted lines represent the per-port (left) and completely shared (right) buffer capacities of the hypothetical switches, adjusted on a per-unit basis to the port radix of our simulation methodology (§6.2). In practice, the buffer is neither partitioned nor fully shared; however, between the two extremes, hardware trends are constraining the throughput-buffering trade-off.

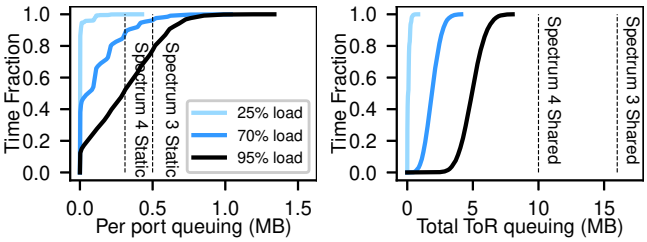


Figure 1: Homa queuing CDFs under various network loads for workload Websearch [10]. The dotted lines represent the switch buffer size adjusted to the actual radix of our simulated ToR; see §6.2.

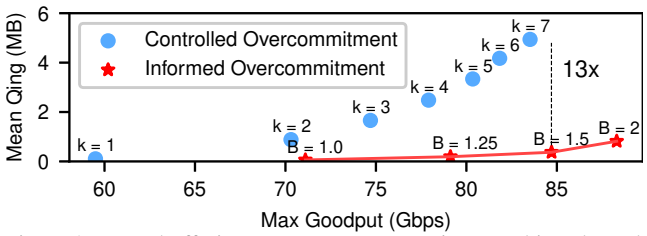


Figure 2: Mean buffering at ToRs versus maximum achieved goodput when sweeping the overcommitment parameter for SIRD (informed overcommitment) and Homa (controlled overcommitment). Results obtained in simulation by running the Websearch workload at 95Gbps on 100Gbps links across 144 servers; see §6.2.

3 SIRD Design Pillars

SIRD is an end-to-end receiver-driven scheme that manages exclusive links proactively and shared links reactively. In practice, this means that receivers perform precise credit-based scheduling when the bottleneck is their downlinks and use congestion feedback to coordinate over shared links. SIRD does not face the limitations of prior work as it:

- is end-to-end, with all decision-making happening at end hosts, and does not rely on advanced switch features or configuration.
- achieves high throughput by using sender and core link congestion feedback to allow receivers to direct credit to links with spare capacity.
- causes minimal buffering in the network by drastically reducing the need for downlink overcommitment.
- does not need switch priority queues to deliver messages with low latency, thanks to minimal buffering.
- can start message transmission immediately without a prior matching stage.
- explicitly tackles core congestion through its shared link management approach.

Efficient credit allocation: SIRD's design allows receivers to send the appropriate amount of credit depending on the real-time availability of the bottleneck link. For downlinks the task is trivial as the link's capacity is managed by one receiver. However, when the bottleneck is shared (sender uplink or core link), SIRD receivers detect competition for bandwidth and adjust the amount of issued credit dynamically. For example, if a sender is the bottleneck for two receivers, each will allocate the appropriate amount of credit for their share of the sender's bandwidth. This approach makes the distribution of credit efficient because it is given to hosts that can promptly use it rather than being accumulated by congested senders. As a result, SIRD can achieve high link utilization using a small amount of credit, *i.e.,* with little overcommitment.

SIRD combines efficient credit allocation and overcommitment in *informed overcommitment*. Each receiver is allotted a limited amount of available credit B. The minimum valid value of B is $1 \times BDP$ as this is the amount of credit required to pull $1 \times BDP$ of inbound traffic and fully utilize the downlink. Higher values of B lead to downlink overcommitment, which increases tolerance to congested shared links but also increases expected queuing.

Figure 2 highlights the benefit of efficient credit allocation in SIRD by comparing the throughput-buffering trade-off across equivalent overcommitment levels for Homa (k), which introduced and uses RD overcommitment, and SIRD (B) under maximum network load. Informed overcommitment yields the same overall goodput despite overcommitting downlinks $14\times$ less, which leads to $13\times$ lower queuing. The goodput-queuing trade-off benefit is maximum at high network load, under which congestion and loss are most likely, and naturally fades as load lessens.

SIRD implements informed overcommitment through a control loop at each receiver, which dynamically adjusts credit allocation across senders based on sender and network feedback. When a sender is concurrently credited by multiple receivers, the sender receives credit faster than it can consume it, causing it to accumulate. This is undesirable as receivers have a limited amount of credit to distribute. Equivalently, limiting accumulation makes credit available to other senders that can actually use it. SIRD's control loop rebalances credit by setting an accumulated credit threshold, $SThr$, that is similar in spirit to DCTCP's marking threshold [8], Swift's target delay [44], or HPCC's η [47]. At each sender, while the amount of accumulated credit from all receivers exceeds $SThr$, a bit is set in all outgoing data packets. Based on arriving bit values, each receiver adjusts the maximum amount of credit that can be allocated to the congested sender in an additive-increase/multiplicative-decrease (AIMD) manner. In this paper, we use DCTCP's [8] AIMD algorithm.

SIRD uses the same mechanisms to handle core congestion with ECN as the signal. Receivers read the CE bit of data packets and adjust per-sender credit limits using AIMD.

Each receiver runs two AIMD algorithms in parallel, one for senders and one for the core, each with its own input signal. The algorithm instance of the most congested area determines credit allocation. This separation of signals allows SIRD to use different signal/algorithm combinations to handle bot-

tlenecks at hosts and switches. Beyond ECN, SIRD can use signals such as end-to-end delay on infrastructures with timestamping support [44] or In-Band Telemetry [47, 69]. In this paper, we use the same proven and simple combination for both and leave the exploration of more complex algorithms and signals to future work.

Starting at line rate: Prior work [31, 36, 39, 56] has demonstrated the latency benefits of starting transmission at line rate, without a gradual ramp-up or preceding control handshakes that take at least one RTT. SIRD's credit allocation mechanism operates *while transmission progress is made*, and thus, senders immediately start at line rate, sending the first BDP bytes *unscheduled* (without waiting for credit).

We further optimize the design based on the following simple observation: small messages benefit the most from unscheduled transmission since their latency is primarily determined by the RTT while throughput-dominated messages see minimal gain. For example, in the absence of queuing, delaying transmission by one RTT increases the end-to-end latency of a message sized at $10 \times BDP$ by 9% compared to 200% for a single-packet message. Therefore, to reduce unnecessary bursty traffic and queuing, SIRD senders do not start transmitting messages larger than a configurable threshold (UnschT) before explicitly receiving credits from the receiver. For messages smaller than UnschT, senders send the first $min(BDP, msg\_size)$ bytes without waiting for credit.

Switch priority queues not required: By efficiently allocating credit, SIRD achieves high link utilization with minimal fabric queuing, thereby largely eliminating its impact on latency. Thus, SIRD can be deployed without any in-network QoS support, though it can benefit in terms of tail latency from having a total of two priority levels.

4 SIRD Design

At a high level, SIRD is an RPC-oriented protocol, similar to other recently proposed datacenter transports [16, 31, 56]. SIRD may be used to implement one-way messages or remote procedure calls [42, 43, 56]. SIRD assumes that the length of each message is known or that data streams will be chunked into messages. SIRD further assumes that ECN is configured in all network switches, with the ECN threshold set according to DCTCP best practices [8]. SIRD is designed to be layered on top of UDP/IP for compatibility with all network deployments. The UDP source port is randomly selected for each packet for fine-grain load balancing, allowing an ECMP network to behave as effectively as random packet spraying [26]. We make no assumption of lossless delivery, in-network priority queues, or smart NICs and switches.

SIRD defines two main packet types: **(1)** `DATA`: a packet that contains part of the payload of a message. Data packets can be scheduled, requiring credit to be sent, or unscheduled. Messages with size $> UnschT$ are scheduled and make an initial credit request by sending a zero-length `DATA` packet; **(2)** `CREDIT`: a control packet sent by the receiver to the sender to schedule the transmission of scheduled `DATA`.

Generally, the flow of packets consists of credit flowing from receivers to senders and data flowing in the opposite direction. Credit leaves the receiver in a `CREDIT` packet, is used by the sender, and returns with a scheduled `DATA` packet.

4.1 Credit Management

Each SIRD receiver maintains two types of credit buckets that limit the amount of credit it can distribute: a global credit bucket and per-sender credit buckets.

The *global credit bucket B* controls overcommitment by capping the total number of outstanding credits that a receiver can issue. Configuring $B \geq BDP$ is necessary to enable link saturation. Further, B bounds the queuing length from scheduled packets to $B - BDP$ bytes at the ToR's downlink. Our analysis (§4.2) and evaluation (§6) show that setting B as low as $1.5 \times BDP$ is sufficient for high link utilization.

Each receiver also maintains a credit bucket per sender machine it communicates with. The *per-sender credit bucket* caps the number of outstanding credits the receiver can issue to a sender. Informed overcommitment is implemented by adjusting the size of the per-sender credit bucket according to the level of congestion at the core and the sender (max $1 \times BDP$). Reducing the bucket size means that less of the receiver's total credit can be allocated to a congested source or path, and thus, more is available for other senders.

4.2 Informed Overcommitment

Informed overcommitment uses two input signals that communicate the extent of congestion in the network and at senders. The signals are carried in `DATA` packets and are the ECN bit in the IP header set by the network and a bit in the SIRD header set by the sender. Each receiver runs two separate AIMD (additive-increase, multiplicative decrease) control loops and uses the most conservative of the two (similar to Swift [44]) to adjust per-sender credit bucket sizes. Each control loop is configured with its own marking threshold ($NThr$ and $SThr$). $NThr$ should be set according to DCTCP best practices [8] to limit queuing in the network core. Note that $NThr$ is much higher than the maximum allowed $B - BDP$ of persistent queueing at ToR switches. Thus, ToR switches never have to mark ECN.

$SThr$ should be set to limit the amount of credit a sender can accumulate, thus allowing receivers to efficiently distribute their limited aggregate credit. Intuitively, $SThr$ determines the level of accumulated credit the control loop is targeting when a sender is congested, *i.e.,* receiving credit faster than it can use it. Setting it too high means each sender can accumulate substantial amounts of credit, and consequently, B would

need to be configured higher to increase aggregate credit availability. Conversely, setting $SThr$ too low does not allow the control loop any slack when converging to a stable state and can cause throughput loss.

To understand the relationship between $SThr$ and B, we analytically examine a simple congested-sender scenario from the perspective of a receiver R to find: how much total credit (B) does R need when receiving from k congested senders, to still have enough credit available to saturate its downlink. Assuming each congested sender sends to f receivers in total, its available uplink bandwidth for R is BW/f. Assuming uniform link speeds, we are interested in the case where senders are the bottleneck and cannot saturate R, or:

$$BW_{supply\_k} < BW_{demand\_R} => \frac{BW}{f} < \frac{BW}{k} => f > k \quad (1)$$

In this case, each of the k senders accumulates up to $SThr$ credit in stable state, or $SThr/f$ from each receiver assuming an equal split (see §4.4). Therefore, to be able to saturate its downlink, R's B must be large enough to allow $1xBDP$ of credit to be in flight despite accumulation at congested senders, or:

$$B \geq BDP + \sum_{k} \frac{SThr}{f} \; ; \; f \geq 2, f > k \quad (2)$$

If k_{max} is the number of congested senders that maximizes the right side of the inequality, then $f_{max} = k_{max} + 1$ (f in denominator) and the maximum value of the term is:

$$\sum_{k}^{k_{max}} \frac{SThr}{k_{max}+1} = \frac{SThr}{k_{max}+1} \sum_{k}^{k_{max}} 1 = SThr \frac{k_{max}}{k_{max}+1} < SThr \quad (3)$$

It follows that, in steady state, R can account for any number of congested senders as long as $B \geq BDP + SThr$. Under dynamic traffic patterns (see §6.2.4) higher values of B can help increase the supply of credit in transient phases. Further, policies other than fair sharing can loosen this property. For example, if senders strictly prioritize some receivers, then, in the worst case where R is de-prioritized by all k senders, Equation 2 loses f from the denominator and B depends on the number of worst-case congested senders.

Table 1: Core configuration parameters.

UnschT	Messages that exceed UnschT in size ask for credit before transmitting.
B	Per-receiver global credit bucket size. Caps credited-but-not-received bytes.
NThr	ECN threshold, configured as for DCTCP.
SThr	Sender marking threshold (*sird.csn*).

Algorithm 1 Receiver Logic

Variables:
- $\boldsymbol{b}$: consumed credit from global receiver bucket of size B,
- $\boldsymbol{sb_i}$: consumed credit from the bucket of sender i,
- $\boldsymbol{senderBkt_i, netBkt_i}$: Sender and network credit bucket size,
- $\boldsymbol{rem_i}$: Requested but not granted credit for sender i.

1: **procedure** onDataPacket(pkt, i) ▷ i:sender
2: $credit \leftarrow getCredit(pkt)$
3: $b \leftarrow b - credit$
4: $sb_i \leftarrow sb_i - credit$
5: $senderBkt_i \leftarrow Sender_{AIMD}(senderBkt_i, pkt.sird.csn)$
6: $netBkt_i \leftarrow Net_{AIMD}(netBkt_i, pkt.ip.ecn)$
7: **end procedure**
8: **procedure** onSendCreditTick() ▷ Runs when $b + min(rem_i, MSS) \leq B$
9: $senderList \leftarrow activeSenders.filter($
$sb_i + min(rem_i, MSS) \leq min(senderBkt_i, netBkt_i))$
10: $s \leftarrow policySelect(senderList)$
11: $credit \leftarrow min(rem_i, MSS)$
12: $sendCredit(s, credit)$
13: $b \leftarrow b + credit$; $sb_i \leftarrow sb_i + credit$; $rem_i \leftarrow rem_i - credit$
14: **end procedure**

Algorithm 2 Sender Logic

Variables: $\boldsymbol{c_r}$: available credit for outbound messages to receiver r

1: **procedure** onCreditPacket($r, credit$)
2: $c_r \leftarrow c_r + credit$
3: **end procedure**
4: **procedure** sendData()
5: $rcvrList \leftarrow activeReceivers.filter(c_r > 0)$;
6: $(r, dataPkt) \leftarrow policySelect(rcvrList)$
7: $dataPkt.sird.csn \leftarrow (\sum_i (c_i) \geq SThr)$
8: $c_r \leftarrow c_r - dataPkt.size$
9: $send(r, dataPkt)$
10: **end procedure**

4.3 Congestion Control Algorithm

Algorithm 1 describes the behavior of a SIRD receiver. When credit is available in the global bucket and on the command of the credit pacer (§4.4), the receiver tries to allocate credit to an active sender (ln. 8). It first selects one of the senders with available credit in the per-sender bucket (ln.9) based on policy (ln.10). The receiver sends a `CREDIT` packet (ln. 12) and reduces the available credit in the global and per-sender buckets, and updates the remaining required credit for that message (ln. 13). Whenever a `DATA` packet arrives (ln. 1), if it is scheduled, the receiver replenishes credit in the global bucket (ln. 3) and the per-sender bucket (ln. 4). Then, it executes the two independent AIMD control loops to adapt to a congested sender (ln. 5) and a congested core network (ln. 6), respectively. The receiver sets the size of the per-sender bucket as the minimum of the two values (ln. 9).

Algorithm 2 describes the implementation of the sender-side algorithm for scheduled `DATA` packets. A host can send

data to a receiver only if it has credit from said receiver. Congested senders mark the *congested sender notification (sird.csn)* bit if the total amount of accumulated credit exceeds $SThr$ (ln. 7). The sender can send unscheduled `DATA` packets at any point in time. It is worth noting that SIRD senders naturally handle scenarios where a meaningful portion of uplink bandwidth is consumed by unscheduled packets. Since senders limit credit accumulation by informing receivers to allocate less credit, the transmission rate of scheduled packets converges to the leftover bandwidth.

4.4 Other Design Concerns

SIRD can be configured to schedule for fairness *e.g.,* by crediting messages in a round-robin manner, or for latency minimization, *e.g.,* by crediting smaller messages first (SRPT), or to accommodate different tenant classes. SIRD implements policies at the receiver (ln .10), which is the primary enforcer, and at the sender (ln. 6). By minimizing queuing in the fabric, SIRD does not need to enforce policies there, simplifying the design. Regardless of which policy is configured at senders, SIRD allocates part of the uplink bandwidth fairly across active receivers, as to ensure a regular flow of congestion information between sender-receiver pairs.

SIRD receivers pace credit transmission to match their downlink's capacity. Pacing improves message latency by reducing downlink queuing from scheduled packets even below the tight $B - BDP$ bound but is not needed for correctness.

If switch priority queues are available, `CREDIT` packets are sent over a higher priority lane to further reduce RTT jitter. The unscheduled prefixes of messages smaller than $UnschT$ also use this lane to bypass transient network queueing. SIRD does not require in-network priority queues to deliver high performance but sees tail latency benefits in some cases (§6.2.4).

Packet loss in SIRD will be very rare by design, but the protocol must still operate correctly in the presence of CRC errors or packet drops due to faults or restarts. SIRD employs Homa's [56] retransmission design in which receivers infer loss when no new packets are received for an incomplete message after a period of a few milliseconds. Upon detecting the loss of a scheduled segment, receivers also reclaim the credit allocated for said segment. SIRD's modest credit overcommitment (*e.g.,* 50%) ensures that receivers can continue fully utilizing their bandwidth even in the improbable event where a sizeable chunk of packets is lost. Finally, SIRD incurs no performance penalties from out-of-order packet deliveries as it does not rely on packet order for loss detection and only delivers completed messages to applications.

5 Caladan Implementation

SIRD, with its message-level API and simple credit management design, can be implemented in NIC hardware or in software. The former eliminates PCIe latencies from the RTT, reducing the *BDP*, and allows for nanosecond-scale pacing and delivery of packets on the wire. We demonstrate that the latter is also possible, at 100Gbps, by implementing SIRD in 4300 SLOC as a UDP-based transport protocol in Caladan [29], a state-of-the-art kernel-bypass dataplane operating system and CPU scheduler. We chose Caladan because of its mature network stack and its lightweight green thread abstraction, suitable for building latency-sensitive applications. SIRD is open source and available on GitHub [1].

The SIRD stack follows an asymmetric design where dedicated spinning threads perform specialized tasks. On the receiver-side, each SIRD node has one dedicated thread that runs the control loops and implements downlink scheduling policies, like round-robin or SRPT, by transmitting paced `CREDIT`s according to Algorithm 1. SIRD reduces network queuing by pacing credit at slightly less than the line rate, as proposed in Hull [9]. Dedicated receiver cores read from RX rings and adjust credits, ensuring the system can quickly replenish new credits for redistribution. Each dedicated receiver core has two RX rings, one for packets of scheduled messages and one for packets of unscheduled messages; this helps avoid drops of scheduled packets during high rates of unscheduled traffic. For efficiency, and to sustain 100Gbps line rates, memory allocation and data copying operations are removed from the credit replenishment path to a small set of dedicated threads which copy packets, reassemble messages from packets, which can be received out of order, and deliver messages to applications via inline callbacks.

The sender-side implementation also has a central thread, which accepts `CREDIT` packets and schedules the transmission of all `DATA` packets, whether these are unscheduled or scheduled. This thread implements Algorithm 2 which marks the *sird.csn* bit of `DATA` packets by comparing the amount of accumulated credit to $SThr$. Like the central receiver thread, this thread paces itself to ensure that packets are not given to the NIC faster than it can consume them. The thread accepts new message requests via asynchronous shared-memory queues, where the application API returns error codes if resource limits are exceeded.

6 Evaluation

We evaluate SIRD using our Caladan implementation and through simulation to answer the following questions: **(1)** Can a software implementation of SIRD deliver 100Gbps along with minimal queuing under incast (§6.1.1) and efficient credit management under outcast (§6.1.2)? **(2)** How well does SIRD navigate the throughput-buffering-latency trade-off in larger scale workloads compared to existing work (§6.2.1)? **(3)** Is SIRD's congestion response robust at high load pressure and can it handle core congestion (§6.2.2)? **(4)** Can SIRD deliver messages with low latency in large scale workloads (§6.2.3)? **(5)** How important is SIRD's sender-informed design in maximizing utilization with minimal buffering? How sensitive is

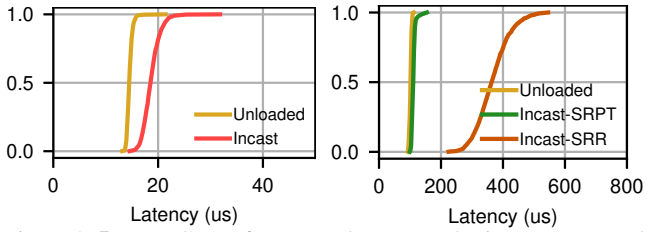


Figure 3: **Incast:** CDF of message latency under incast compared to an unloaded baseline for 8B requests (left) and 500KB requests (right). The incast is formed by six senders transmitting 10MB messages in an open loop.

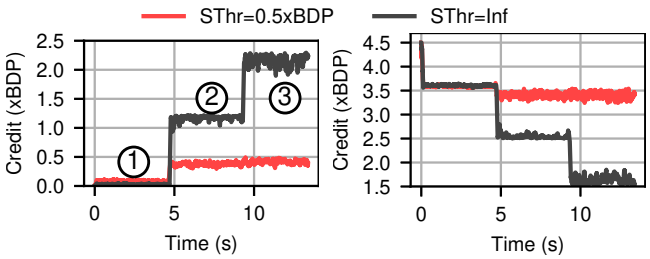


Figure 4: Left: credit accumulated at congested sender. Right: sum of credit available at the three receivers - initial total: $4 \times 1.5 = 4.5 \times BDP$. 100ms moving average. The circled numbers indicate the number of receivers at that stage of the experiment.

SIRD to its parameters (§6.2.4)?

6.1 System Evaluation

We evaluate SIRD at a small scale using Cloudlab's [27] sm110p machines equipped with 100Gbps ConnectX-6 DX NICs and Xeon 4314 CPUs, all located in the same rack. Our implementation uses 9KB jumbo frames and the unloaded RTT of the SIRD-Caladan stack is approximately 18 µs. Latency measurements are end-to-end, measured by the client, and include four request/reply copies to/from the application layer. We sample other system metrics every 2ms. While the size of requests may vary from experiment to experiment, replies are of minimal size.

We configure SIRD parameters as follows: $BDP = 216KB$ (24 jumbo frames), $B = 1.5 \times BDP$, $SThr = 0.5 \times BDP$, $UnschT = 1.0 \times BDP$. We do not use switch priority queues.

6.1.1 Receiver Congestion

SIRD's main objectives when handling incast are eliminating queuing-induced latency for small messages and being able to reduce latency for larger messages through credit-based scheduling. To evaluate this, we run an experiment where six senders saturate a receiver by sending 10MB requests at a rate of 17Gbps each. A seventh sender periodically transmits either 8B or 500KB requests and captures the latency distribution.

Figure 3 (left) plots the round-trip latency experienced by short requests, which are unscheduled. When the receiver is saturated, we observe only a few microseconds of additional latency compared to when the receiver is unloaded, which corresponds to a couple of packets of queuing, either at the ToR or at the host stack. For context, the median latency under load for the same experiment using kernel TCP Cubic was above 1ms. SIRD achieves this result because it limits the number of scheduled inbound bytes to B and thus the extent of downlink queuing to $B - BDP$. Further, since in this case senders respond to credit immediately, credit pacing effectively eliminates downlink queuing, while still achieving 96Gbps in this experiment.

Figure 3 (right) shows the latency of *scheduled* 500KB requests under two receiver scheduling policies. Under SRPT, the receiver allocates credit to the message with the fewest remaining bytes and thus quickly prioritizes 500KB over 10MB requests, leading to near-unloaded latency despite the saturated downlink. SIRD can also implement fairer policies like per-sender round robin ("SRR") as shown. Finally, as it causes minimal network queuing, SIRD can implement such policies without in-network QoS support.

6.1.2 Sender Information

When senders are the bottleneck (outcast), SIRD tackles the congested sender problem by providing congestion feedback to receivers and allowing them to adjust their credit allocation. To evaluate the mechanism's efficacy at 100Gbps, we run an outcast experiment where a single sender sends 10MB messages at full rate to three receivers in a time-staggered manner. Figure 4 (left) shows that, without informed overcommitment ("$SThr = Inf$"), the level of credit accumulated at the congested sender increases with every new receiver. This is because each receiver independently gives the sender $1 \times BDP$ worth of credit to achieve full rate. Through congestion feedback ("$SThr = 0.5 \times BDP$"), receivers coordinate and scale down their credit allocation until credit accumulation at the sender is less than $SThr = 0.5 \times BDP$, on average. Credit, instead, stays at receivers, and can be used to schedule other senders. Figure 4 (right) shows the sum of credit available at the three receivers, each receiving a max-min fair share of the sender's bandwidth. Without informed overcommitment, each receiver allocates $1 \times BDP$ worth of credit to the congested sender. With it, each allocates $(BDP + SThr)/num\_rcvers$.

6.2 Simulations

We use network simulation to evaluate SIRD on large-scale workloads and compare it to other protocol designs.

We compare SIRD to 5 baselines: DCTCP [8], a widely deployed sender-driven scheme [33], Swift [44], a state-of-the-art production sender-driven scheme, Homa [56], because of its near-optimal latency and its use of overcommitment, ExpressPass [20], as it employs a hop-by-hop approach to managing credit, and dcPIM [16], as a unique point in the

design space as it explicitly matches senders and receivers. We do not extend Homa with Aeolus [39] for the reasons discussed in [40]. Note that the Homa and dcPIM papers [16,56] already include favorable comparisons to NDP [36], Aeolus [39], PIAS [13], pHost [31], and HPCC [47], thus we did not include them in our evaluation.

We implement SIRD on ns-2 [11], reuse the original ns-2 DCTCP implementation (using the same parameters as [10], scaled to 100Gbps) and the original ns-2 ExpressPass implementation [21], and port the published Homa simulator [55] to ns-2 using the same parameters as in [56] scaled to 100Gbps. The published Homa simulator does not implement the incast optimization [56], which further relies on two-way messages. We use a community Swift simulator, kindly made public [3], and configure its delay parameters to achieve similar throughput to DCTCP, respecting the guidelines [44]. Finally, we use the published dcPIM simulator [15]. SIRD's simulator is open source and available on GitHub [2].

Table 2 lists protocol parameter values. DCTCP and Swift use pools of pre-established connections (40 for each host pair). SIRD approximates SRPT scheduling like Homa and dcPIM. SIRD, Homa, and dcPIM use packet spraying, DCTCP and Swift use ECMP. Homa uses 8 network priority levels, dcPIM 3, and SIRD 2 (§4.4). The initial window of DCTCP and Swift is configured at *BDP* like in [10]. The RTT latency is similar to prior work.

Topology: We simulate the two-tier leaf-spine topology used in previous work [8, 10, 16, 31, 56] with 144 hosts, connected to 9 top-of-rack switches (16 hosts each) via 4 spine switches, with link speeds of 100Gbps to hosts and 400Gbps to spines. We simulate switches with infinite buffers, *i.e.,* without packet drops (1) to avoid making methodologically complex assumptions as drop rates and thus latency and throughput are very sensitive to switch buffer sizes, organization, and configuration [33,40] (see §2.2); and (2) to study the intended mode of operation of the protocols that leverage buffering to achieve high link utilization and operate best without drops. The intentional dropping of credit packets to adjust sender rates by ExpressPass is maintained. SIRD never uses more than a small fraction of the theoretical capacity (§6.2.1) and is not affected by this setup.

Workload: Each host operates both as client and server, sending one-way messages according to an open-loop Poisson distribution to uniformly random receivers (all-to-all). We simulate 3 workloads: **(1) WKa**: an aggregate of RPC sizes at a Google datacenter [28]; **(2) WKb**: a Hadoop workload at Facebook [64]; **(3) WKc**: a web search application [10]. We select them to test over a wide range of mean message sizes of 3KB, 125KB, and 2.5MB respectively.

We simulate these 3 workloads on 3 traffic configurations, for a total of 9 points of comparison. The configurations are: **(1) Balanced:** The default configuration described above. We vary the applied load, which does not include protocol-dependent header overheads, from 25% to 95% of link capacity. **(2) Core:** Same as (1) but ToR-Spine links are 200Gbps (2-to-1 oversubscription). Due to uniform message target selection, $128/144 \approx 89\%$ of messages travel via spines, turning the core into the bottleneck. We consequently reduce the load applied by hosts by $\times 0.89 * 2$ to reflect the network's reduced capacity. This is not meant to reflect a permanent load distribution, but we hypothesize that it is possible transiently. **(3) Incast:** We use the methodology of [16,47] and combine background traffic with overlay incast traffic: 30 random senders periodically send a 500KB message to a random receiver. Incast traffic represents 7% of the total load.

We report goodput (rate of received application payload), total buffering in switches, and message slowdown, defined as the ratio between the measured and the minimum possible latency for each message. In the incast configuration, we exclude incast messages from slowdown results.

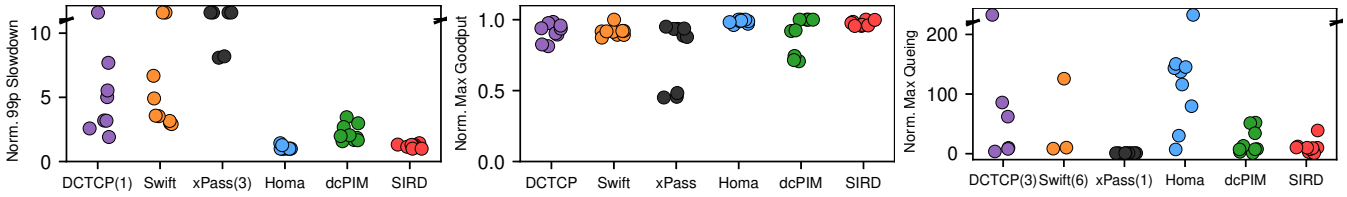
6.2.1 Performance Overview

Figure 5 shows how protocols navigate the trade-offs between throughput, buffering, and latency by plotting their relative performance across all 9 workload-configuration combinations. The best-performing protocol on each of the 9 scenarios gets a score of 1.0 (per metric) and the others are normalized to it, so that goodput is always ≤ 1.0 whereas queuing and slowdown are always ≥ 1.0. We report the highest achieved goodput and peak queuing over all the levels of applied load. We report 99^{th} percentile slowdown across all messages of each workload at 50% applied load which is a level most protocols can deliver in all scenarios. The following figures and calculations do not include cases where a protocol cannot satisfy a specific load level or cannot stop network buffers from growing infinitely.

Overall, SIRD is the only protocol that consistently achieves near-ideal scores across all metrics. Specifically, SIRD causes 12× less peak network queuing than Homa and achieves competitive latency and goodput performance.

Table 2: Default simulation parameters for each protocol.

Prot.	Parameters
all	•RTT(MSS): 5.5µs intra-rack, 7.5µs inter-rack •$BDP = 100KB$; link@100Gbps
SIRD	•B : $1.5 \times BDP$, •$UnschT$: $1 \times BDP$ •$NThr = 1.25 \times BDP$, •$SThr = 0.5 \times BDP$
DCTCP	•Initial window: $1 \times BDP$, •g=0.08 •Marking ECN Threshold: $1.25 \times BDP$
Swift	•Initial window: $1 \times BDP$ •base_target: $2 \times RTT$, •fs_range: $5 \times RTT$ •$\hbar$: $1.25 \times BDP$, •fs_max: 100, •fs_min: 0.1
XPass	•$\alpha = 1/16$ •$w_{init} = 1/16$ •$loss\_tgt = 1/8$
Homa	Same as [56] at 100Gbps (incl. priority split)
dcPIM	Same as [16]



(a) Slowdown (b) Goodput (c) Queuing

Figure 5: Normalized goodput, queuing, and slowdown across all 9 configurations. Each metric is normalized based on the best-performing protocol for the given metric and configuration. For queuing and slowdown, lower is better. For goodput, higher is better. Normalized slowdown and buffering are capped at 10× and 200×, respectively, and higher values are plotted in the overflow area. The numbers in parentheses show the number of unstable configurations for each protocol which are not plotted. X-axis jitter is added for visibility. Find the data in Table 4.

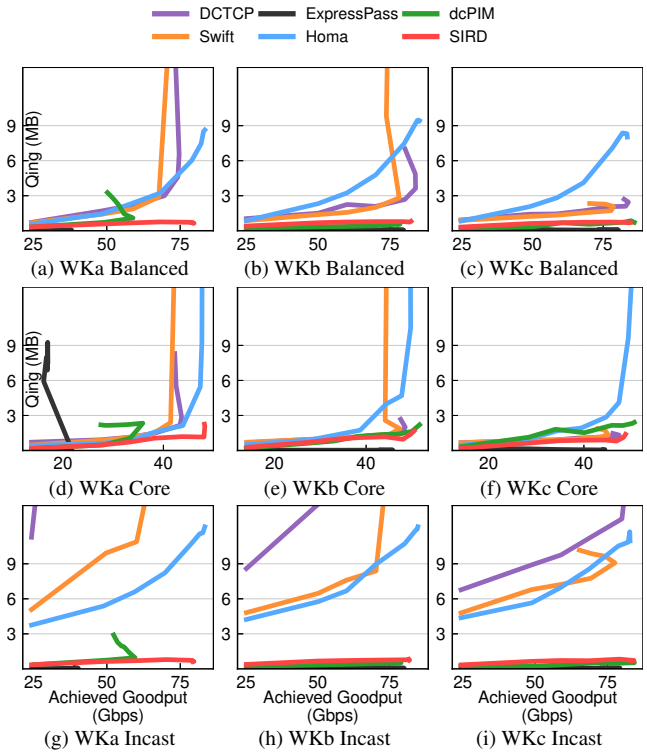
SIRD outperforms dcPIM in message slowdown, and peak goodput and queuing by 46%, 9%, and 43% respectively. ExpressPass causes practically zero queuing thanks to its hop-by-hop design, and 88% less than SIRD, but SIRD delivers 10× lower slowdown and 26% more goodput. Even under full fabric saturation, SIRD induces at most 0.8MB of ToR queuing in receiver-bottleneck scenarios and 2.3MB in core-bottleneck scenarios. Given a packet buffer capacity of 3.13MB/Tbps (see §2.2), these values correspond to a maximum buffer occupancy of 8% and 23%, respectively.

6.2.2 Congestion Response

We now zoom in on how each protocol manages congestion across various levels of applied stress. Figure 6 plots maximum buffering across ToR switches as a function of achieved goodput for balanced (top), core (middle), and incast (bottom) configurations. The reported goodput is the mean across all 144 hosts and reflects the rate of message delivery to applications. ToR queuing covers both downlinks and links to aggregation switches (core).

Overall, SIRD consistently achieves high goodput while causing minimal buffering across load levels, even under high levels of stress. Through informed overcommitment, SIRD makes effective use of a limited amount of credit and achieves high goodput while reducing the need for packet buffer space by up to 20× compared to Homa's controlled overcommitment. Further, SIRD displays a more predictable congestion response compared to dcPIM and ExpressPass. As expected, based on Figure 5a, DCTCP and Swift cause meaningful buffering without achieving exceptional goodput. Mean queuing is qualitatively similar (appendix Figure 13).

dcPIM and ExpressPass sometimes compare favorably to SIRD but only when the mean message size is not small. They both struggle in terms of achieved goodput and buffering in **WKa** for all three configurations (left column). In WKa, ≈ 99% of messages are smaller than $1 \times BDP$ and are responsible for ≈ 40% of the traffic. ExpressPass's behavior is known and discussed in [20]: more credit than needed may be sent for a small message which may then compete for bandwidth with productive credit of a large message. For dcPIM, the likely reason as discussed in [41] is that to send a small

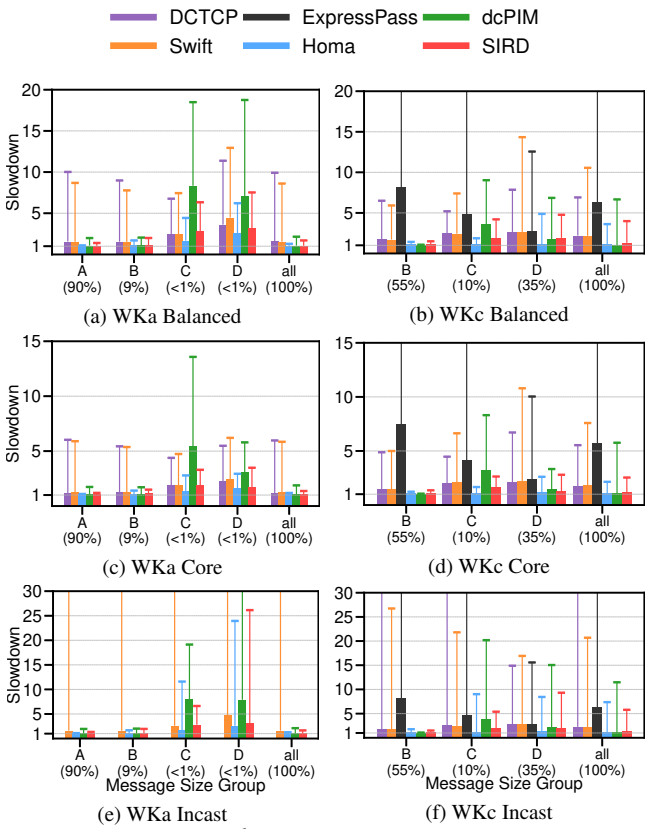


(a) WKa Balanced (b) WKb Balanced (c) WKc Balanced
(d) WKa Core (e) WKb Core (f) WKc Core
(g) WKa Incast (h) WKb Incast (i) WKc Incast

Figure 6: Maximum ToR queuing vs. achieved goodput. Configurations: Balanced (top), Core (middle), Incast (bottom).

message, a sender has to preempt the transmission of a larger message which can cause the receiver of the latter to remain partially idle. SIRD's behavior is consistent across workload message sizes thanks to slight downlink overcommitment which absorbs discontinuities in large message transmission.

In the core configuration (Figure 6 - middle row), where the core is the bottleneck, we observe that SIRD's reactive congestion management, despite sharing the same ECN-based mechanism as DCTCP, achieves steadier behavior because it limits the number of bytes in the network. Homa, though lacking an explicit mechanism to regulate queing at the core, does constrain peak queuing below 19MB (6MB/Tbps of switch BW) because it limits outstanding bytes to the aggregate overcommitment level of receivers. The same applies to dcPIM



(a) WKa Balanced (b) WKc Balanced

(c) WKa Core (d) WKc Core

(e) WKa Incast (f) WKc Incast

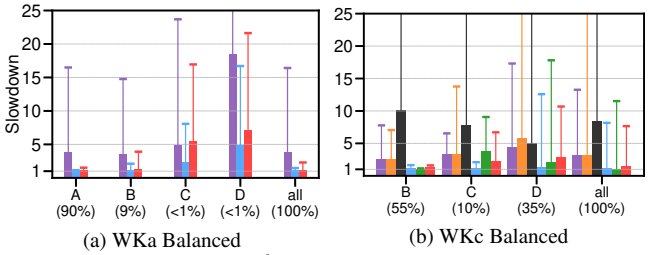
Figure 7: Median and 99^{th} percentile slowdown at 50% offered application load. Each bar group contains the messages in the following size ranges: $0 \leq A < MSS \leq B < 1 \times BDP \leq C < 8 \times BDP \leq D$. Also shown is the percentage of messages that belong to each group. WKc has no sub-MSS messages. Protocols that cannot deliver 50% load are not shown.

though queuing is much lower as it does not overcommit.

Last, the incast configuration (bottom row) illustrates the relative advantage of RD schemes over DCTCP and Swift. Homa's results would be different if the incast optimization described in [56] was implemented and the workload consisted of request-response pairs instead of one-way messages.

6.2.3 Message Latency

Figure 7 shows the median and 99^{th} percentile slowdown for different message size ranges across configurations at 50% applied load (except WKb results which fall between the other two and can be found in appendix Figure 12). Across each workload as a whole (rightmost bar cluster "all"), SIRD is generally on par with Homa, and generally outperforms dcPIM in terms of tail latency. For small, latency-sensitive messages $< BDP$ (100KB - groups A and B), SIRD, dcPIM, and Homa offer close to hardware latency. Both in aggregate and for small messages, DCTCP and Swift perform an order of magnitude worse at the tail because they cause meaningful buffering without having a bypass mechanism like Homa.



(a) WKa Balanced (b) WKc Balanced

Figure 8: Median and 99^{th} percentile slowdown at 70% offered application load. Protocols that cannot deliver 70% load are not shown. Legend same as in Figure 7.

For messages larger than $1 \times BDP$ (groups C and D), SIRD comes closest to near-optimal Homa and strongly outperforms the other protocols. SIRD achieves up to $4\times$ lower latency than dcPIM in this size range because it does not wait for multi-RTT handshakes before sending a message as discussed in §3. Note that Figure 7b is comparable to Figure 3d in [16]. Similarly, SIRD outperforms ExpressPass on latency because it does not take multiple RTTs to capture the full link bandwidth, and because it implements SRPT.

The differences in latency for protocols other than dcPIM and ExpressPass, which wait before ramping up transmission, can be explained through their *effective* scheduling policies. That is, assuming that a protocol manages to deliver enough messages to achieve 50% goodput, lowering latency is a matter of appropriately ordering the transmission of messages (assuming no conflicting application concerns). For example, DCTCP achieves equivalent or better latency than Swift in all but the incast scenarios. We argue that this is because Swift is better at fairly sharing bandwidth between messages thanks to its faster-converging control loop. Latency-wise, fair sharing is inferior to SRPT since it delays individual messages in favor of equitable progress. Along the same lines, SIRD cannot always match the latency of Homa because it approximates SRPT less faithfully: (1) unlike Homa, a portion of the sender uplink is fair-shared (50% in this case). (2) To avoid credit accumulation in congested senders, informed overcommitment adjusts per-sender credit bucket sizes equitably. This mostly impacts the latency of group C, as the bucket size may be small due to sender congestion and not grow during the lifetime of a message smaller than $8 \times BDP$. Consequently, compared to near-optimal Homa in the Balanced configuration, SIRD's 99^{th} percentile slowdown in group C is $1.85\times$ and $2.68\times$ higher at 50% and 70% application load respectively, while it outperforms the other protocols.

Figure 8 shows slowdown under a higher applied application load of 70% for the protocols that can deliver it. At this load level, message scheduling becomes more important and Homa's near-optimal SRPT implementation sees its relative performance improve in some cases.

In summary, SIRD delivers messages with low latency, outperforms dcPIM, ExpressPass, DCTCP, and Swift, and is competitive with Homa, which nearly optimally approximates

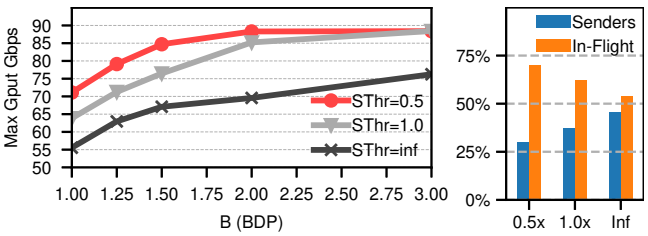


Figure 9: Left: Max goodput across values of B and $SThr$. Right: Credit location at max goodput as a function of $SThr$. $B = 1.5 \times BDP$. "In-flight" is CREDIT and DATA. Credit at receivers is low at this load.

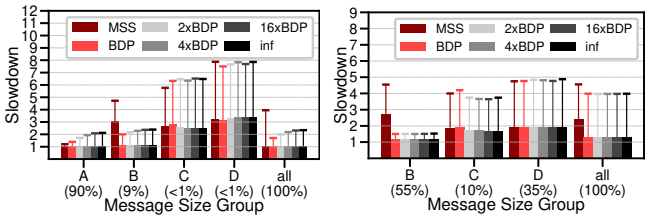


Figure 10: Slowdown as a function of UnschT for WKa (left) and WKc (right) at 50% application load in the balanced configuration;

SRPT [56].

6.2.4 Sensitivity Analysis

In this section, we explore SIRD's sensitivity to its key parameters: B, $SThr$, and $UnschT$, as well as to the availability of switch priority queues. SIRD's sensitivity to $NThr$ is the same as DCTCP's to K [8].

Informed overcommitment: Equation 2 introduced the linear steady-state relationship between B and $SThr$, the two key parameters of informed overcommitment. Figure 9 (left) shows the measured maximum achieved goodput in Balanced WKc as a function of B and $SThr$.

We observe that the presence of informed overcommitment increases achievable goodput by ∼25%, confirming that the introduced sender-informed mechanism is necessary to fully utilize the network with a limited amount of credit, and thus low queuing. When the mechanism is disabled ($SThr = \text{inf}$), there is stranded and unused credit at congested senders, which prevents receivers from achieving the full rate. Figure 9 (right) shows where in the topology credit is under max goodput. Lower $SThr$ values reduce the credit accumulation at senders, and improves throughput by increasing the number of in-flight CREDIT and DATA packets.

With the mechanism enabled, the curves asymptotically converge to the same maximum goodput of 90Gbps. Reaching the plateau with a lower value of B also demands lowering $SThr$ to reduce credit stranded at senders (Equation 2). Queuing increases with B as in Figure 2 and remains stable when varying $SThr$. We selected $B = 1.5 \times BDP$ rather than $B = 2.0 \times BDP$ for our experiments because it halves the maximum buffering caused by scheduled packets from $1\times$ to $0.5 \times BDP$ ($= B - BDP$). We do not configure $SThr$ lower than $0.5 \times BDP$ as we deem it unrealistic to implement in software, as it may cause unwanted marking due to batch credit arrivals. The minimum correct value for this topology is $0.15 \times BDP$, as per the guidelines [8].

Unscheduled transmissions: SIRD allows the unscheduled transmission of the BDP prefix of messages $\leq UnschT$. Figure 10 explores the sensitivity to this parameter using workloads WKa and WKc. Setting $UnschT = MSS$ meaningfully increases median and tail latency for messages with sizes in $[MSS, BDP]$ while higher values offer no appreciable net benefit. This also holds at 70% load. Increasing $UnschT >> BDP$ in WKc has little impact on max and mean ToR buffering, which range within [696,747]KB and [363,378]KB respectively. In contrast, because 99% of messages in WKa are unscheduled, max and mean ToR buffering scale from 501KB to 1016KB and 220KB to 435KB, respectively, as UnschT increases. It is worth noting that saturating the fabric's bandwidth, which is when queuing is maximized, is far less likely with high-request-rate workloads like WKa.

The default value of $UnschT = 1 \times BDP$ is a satisfying compromise as large values do not yield latency benefits yet can unnecessarily expose the fabric to coordinated traffic bursts (*e.g.,* incast). We confirm this by running WKc under the incast configuration which has a high concentration of $5 \times BDP$ message bursts. We observe the following performance degradation when comparing $UnschT = 4 \times BDP$ to $UnschT = 16 \times BDP$: Overall 99^{th} percentile slowdown increases by 34% while maximum and mean ToR queuing increases by 5.7× and 80% respectively. These results justify our decision to introduce a size threshold above which messages are entirely scheduled.

In deployments with severe incast consisting of messages smaller than BDP, UnschT can be configured to be smaller than MSS. If, for example, 100 hosts concurrently send a $0.5 \times BDP$ message to a single receiver, the latter will effectively receive 100 small packets requesting credit. The receiver will immediately start issuing credit based on its policy and ramp-up to full bandwidth utilization one RTT after the senders transmitted. In TCP terms, this specific scenario is roughly equivalent to each sender starting with a window of one MSS, and ramping to the correct window value just one RTT later.

Use of switch priority queues: SIRD may use a second 802.1p priority level for control packets and/or unscheduled DATA. Figure 11 shows how the use of priorities impacts message slowdown for WKa and WKc in terms of (i) possible degradation when switch priority queues are unavailable, and (ii) the benefit of prioritizing unscheduled DATA.

We observe that, across message sizes, median slowdown is largely unaffected and tail slowdown benefits from high priority transmission in some cases. Small messages benefit because they bypass the small queues SIRD forms ($\approx 1 \times BDP$ per link at the 99^{th} percentile and $\approx 0.1 \times BDP$ on average).

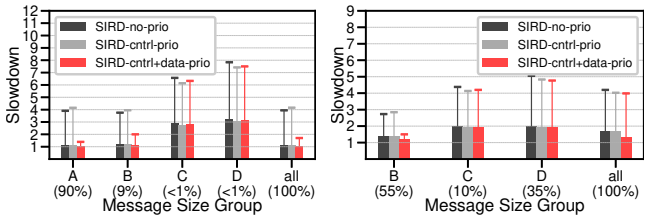


Figure 11: Slowdown as a function of priority use for WKa (left) and WKc (right) at 50% application load in the balanced configuration;

This is more evident in WKa where 99% of messages are unscheduled. Sensitivity to priorities is similar at 70% load (not shown), where, compared to 50% load, group A's tail slowdown benefits 5% more from `DATA` prioritization and group B's 6% less.

Maximum goodput increases by 1% and 2.4% with control packet prioritization for WKa and WKc respectively, as credit is delivered slightly more predictably. Queuing in the network is also insensitive to the use of priorities. Given SIRD's low sensitivity to the availability of switch priority queues, we argue that it can be deployed without this dependency with little sacrifice. This differentiates SIRD from Homa, which is designed around the use of priorities to bypass long in-network queues, and dcPIM, which strongly benefits from tight control packet delivery due to its semi-synchronous nature.

7 Related Work

The topic of datacenter congestion control has been extensively explored due to the increasing IO speeds, the µs-scale latencies, emerging programmable hardware, RPC workloads, and host-centric concerns [4–6, 8–10, 12, 13, 19, 20, 22, 31, 34, 36, 38, 42–48, 50, 51, 54, 56, 57, 61, 66–71, 74].

SD schemes like DCQCN [74], Timely [54], HPCC [47], Bolt [12], and PTT [67] use network signals in the form of ECN [8, 68, 74], delay [44, 54], in-band telemetry [47, 69, 73], and packet drops [10] to adjust sending windows [8, 44, 47, 66] or rates [38, 54, 70]. Some operate end-to-end with no or trivial switch support [8, 44, 54, 67] while others leverage novel but non-universal switch features to improve feedback quality [47, 69, 73] or to tighten reaction times [12, 19, 71].

RD protocols attempt to avoid congestion altogether by proactively scheduling packet transmissions [5, 16, 20, 31, 36, 39, 46, 49, 56, 61] and have already been discussed (§2). Orthogonally to the scope of this paper, FlexPass [49] enables fair bandwidth sharing among proactive and reactive protocols. EQDS [59] enables the coexistence of traditional protocols like DCTCP and RDMA by moving all queuing interactions to end-hosts and using an RD approach to admit packets into the network fabric.

Previous work has followed different approaches regarding the desired flow/message scheduling policy. The traditional approach is to aim for fair bandwidth allocation at the flow level [8, 44, 54, 74]. Alternatively, D^2TCP, D^3, and Karuna [18] prioritize flows based on deadlines communicated by higher-level services. A recent line of work [10, 13, 31, 39, 56], which includes both proactive and reactive protocols, attempts to approximate SRPT or SJF (shorted job first) scheduling in the fabric which minimizes average latency [10, 14]. SIRD can approximate a variety of policies at receivers and senders and avoids the need for fabric policies as it keeps in-network queuing minimal.

8 Conclusion

SIRD takes a holistic approach to datacenter congestion management by tackling the fundamental tension of receiver-driven designs, which is the management of shared links. SIRD applies proactive scheduling to exclusive receiver downlinks, which are statistically the most congested, and leverages sender and fabric congestion feedback to reactively allocate the bandwidth of shared links. Compared to existing designs, SIRD manages to simultaneously deliver high link utilization with hardly any network buffering, and low message latency. SIRD does so without assuming advanced switch ASIC features nor switch priority queues, which makes it compatible with existing and heterogeneous datacenter networks. We implemented a prototype of SIRD in software, on the Caladan stack, and showed that it can efficiently allocate credit while delivering 100Gbps.

Acknowledgements

The authors thank John Ousterhout for numerous discussions on the topic of receiver-driven protocols. We thank our shepherd Srikanth Sundaresan, Nate Foster, Katerina Argyraki, Haitham Al Hassanieh, James Larus, Charly Castes, Neelu Kalani, Boris Pismenny, Rui Yang, Mahyar Emami, Sahand Kashani, Mia Primorac, Francois Costa, and all anonymous reviewers for their insightful comments.

This work was funded in part by the Microsoft-EPFL Joint Research Center and used CloudLab [27] for 100GbE experiments. Ryan Kosta participated in EPFL's Excellence Research Internship Program.

References

[1] SIRD implementation on Caladan. https://github.com/epfl-dcsl/SIRD-Caladan-Impl, 2025.

[2] SIRD simulator repository. https://github.com/epfl-dcsl/SIRD-Simulator, 2025.

[3] Sepehr Abdous, Erfan Sharafzadeh, and Soudeh Ghorbani. Burst-tolerant datacenter networks with Vertigo. In *Proceedings of the 2021 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1–15, 2021.

[4] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–70, 2022.

[5] Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David B. Shmoys, and Amin Vahdat. Harmony: A Congestion-free Datacenter Architecture. In *Proceedings of the 21st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 329–343, 2024.

[6] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[7] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 503–514, 2014.

[8] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[9] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, 2012.

[10] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[11] Eitan Altman and Tania Jimenez. NS simulator for beginners. *Synthesis Lectures on Communication Networks*, 5(1):1–184, 2012.

[12] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 219–236, 2023.

[13] Wei Bai, Kai Chen, Hao Wang, Li Chen, Dongsu Han, and Chen Tian. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 455–468, 2015.

[14] Amotz Bar-Noy, Magnús M. Halldórsson, Guy Kortsarz, Ravit Salman, and Hadas Shachnai. Sum Multicoloring of Graphs. *J. Algorithms*, 37(2):422–450, 2000.

[15] Qizhe Cai. dcpim simulator repository. https://github.com/Terabit-Ethernet/dcPIM/tree/master/simulator, 2023.

[16] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcPIM: near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 53–65, 2022.

[17] Chih-Hao Chang, VS Chang, KH Pan, KT Lai, JH Lu, JA Ng, CY Chen, BF Wu, CJ Lin, CS Liang, et al. Critical process features enabling aggressive contacted gate pitch scaling for 3nm cmos technology and beyond. In *2022 International Electron Devices Meeting (IEDM)*, pages 27–1. IEEE, 2022.

[18] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 174–187, 2016.

[19] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–28, 2014.

[20] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 239–252, 2017.

[21] Inho Cho, Keon Jang, and Dongsu Han. Expresspass simulator repository. https://github.com/kaist-ina/ns2-xpass, 2018.

[22] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for µs-scale RPCs with Breakwater. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 299–314, 2020.

[23] Ultra Ethernet Consortium. Overview of and Motivation for the Forthcoming Ultra Ethernet Consortium Specification. https://ultraethernet.org/wp-content/uploads/sites/20/2023/10/23.07.12-UEC-1.0-Overview-FINAL-WITH-LOGO.pdf, 2023.

[24] David Schor. IEDM 2022: Did We Just Witness The Death Of SRAM? https://fuse.wikichip.org/news/7343/iedm-2022-did-we-just-witness-the-death-of-sram/, 2022.

[25] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[26] Advait Abhay Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *Proceedings of the 2013 IEEE Conference on Computer Communications (INFOCOM)*, pages 2130–2138, 2013.

[27] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[28] Bob Felderman. Personal communication to the authors of [56], 2018.

[29] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 281–297, 2020.

[30] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 249–264, 2016.

[31] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[32] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network Aggregation for Shared Machine Learning Clusters. In *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys)*, 2021.

[33] Ehab Ghabashneh, Yimeng Zhao, Cristian Lumezanu, Neil Spring, Srikanth Sundaresan, and Sanjay Rao. A microscopic view of bursts, buffer contention, and loss in data centers. In *Proceedings of the 22nd ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 567–580, 2022.

[34] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure Flow Control. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 779–805, 2022.

[35] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 51–62, 2009.

[36] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[37] Torsten Hoefler, Duncan Roweth, Keith D. Underwood, Robert Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Moray McLaren, Abdul Kabbani, and Steve Scott. Data Center Ethernet and Remote Direct Memory Access: Issues at Hyperscale. *Computer*, 56(7):67–77, 2023.

[38] Chi-Yao Hong, Matthew Caesar, and Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 127–138, 2012.

[39] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 422–434, 2020.

[40] John K. Ousterhout. Homa Wiki on Aeolus. https://homa-transport.atlassian.net/wiki/spaces/HOMA/pages/262185/A+Critique+of+Aeolus+A+Building+Block+for+Proactive+Transports+in+Datacenters, 2022.

[41] John K. Ousterhout. Homa Wiki on dcPIM. https://homa-transport.atlassian.net/wiki/spaces/HOMA/pages/1507461/A+Critique+of+dcPIM+Near-Optimal+Proactive+Datacenter+Transport, 2022.

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[43] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[44] 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. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, pages 514–528, 2020.

[45] Yanfang Le, Jeongkeun Lee, Jeremias Blendin, Jiayi Chen, Georgios Nikolaidis, Rong Pan, Robert Soulé, Aditya Akella, Pedro Yebenes Segura, Arjun Singhvi, Yuliang Li, Qingkai Meng, Changhoon Kim, and Serhat Arslan. SFC: Near-Source Congestion Signaling and Flow Control. *CoRR*, abs/2305.00538, 2023.

[46] Yanfang Le, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Sujata Banerjee, Aditya Akella, and Michael M. Swift. PL2: Towards Predictable Low Latency in Rack-Scale Networks. *CoRR*, abs/2101.06537, 2021.

[47] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, pages 44–58, 2019.

[48] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of the 2021 EuroSys Conference*, pages 33–48, 2021.

[49] Hwijoon Lim, Jaehong Kim, Inho Cho, Keon Jang, Wei Bai, and Dongsu Han. FlexPass: A Case for Flexible Credit-based Transport for Datacenter Networks. In *Proceedings of the 2023 EuroSys Conference*, pages 606–622, 2023.

[50] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–63, 2021.

[51] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve D. Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[52] Mellanox. Spectrum 2 datasheet. https://nvdam.widen.net/s/gbk7knpsfd/sn3000-series, 2017.

[53] Mellanox. Spectrum 3 datasheet. https://nvdam.widen.net/s/6269c25wv8/nv-spectrum-sn4000-product-brief, 2020.

[54] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 537–550, 2015.

[55] Behnam Montazeri. Homa simulator repository. https://github.com/PlatformLab/HomaSimulation, 2019.

[56] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[57] Mohammad Noormohammadpour and Cauligi S. Raghavendra. Datacenter Traffic Control: Understanding Techniques and Tradeoffs. *IEEE Commun. Surv. Tutorials*, 20(2):1492–1525, 2018.

[58] Nvidia. Spectrum 4 datasheet. https://nvdam.widen.net/s/mmvbnpk8qk/networking-ethernet-switches-sn5000-datasheet-us, 2022.

[59] Vladimir Andrei Olteanu, Haggai Eran, Dragos Dumitrescu, Adrian Popa, Cristi Baciu, Mark Silberstein, Georgios Nikolaidis, Mark Handley, and Costin Raiciu. An edge-queued datagram service for all datacenter traffic. In *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 761–777, 2022.

[60] Amy Ousterhout, Adam Belay, and Irene Zhang. Just In Time Delivery: Leveraging Operating Systems Knowledge for Better Datacenter Congestion Control. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*, 2019.

[61] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 307–318, 2014.

[62] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware Job Scheduling in Machine Learning Clusters. *CoRR*, abs/2308.00852, 2023.

[63] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040.

[64] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.

[65] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 183–197, 2015.

[66] Le Sun, Hai Dong, Omar Khadeer Hussain, Farookh Khadeer Hussain, and Alex X. Liu. A framework of cloud service selection with criteria interactions. *Future Gener. Comput. Syst.*, 94:749–764, 2019.

[67] Lide Suo, Yiren Pang, Wenxin Li, Renjie Pei, Keqiu Li, Xiulong Liu, Xin He, Yitao Hu, and Guyue Liu. PPT: A Pragmatic Transport for Datacenters. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 954–969, 2024.

[68] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-aware datacenter tcp (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 115–126, 2012.

[69] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 255–274, 2023.

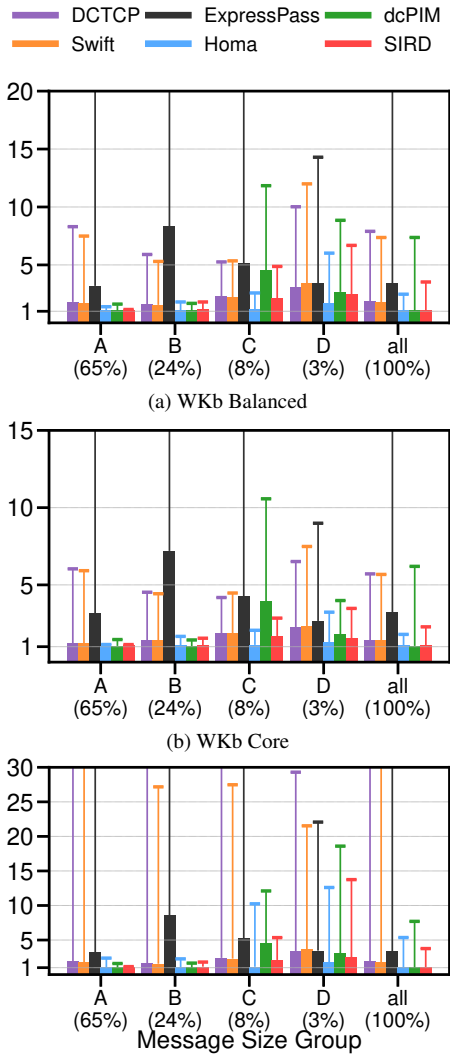
[70] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Antony I. T. Rowstron. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 50–61, 2011.

[71] David Zats, Anand Padmanabha Iyer, Ganesh Ananthanarayanan, Rachit Agarwal, Randy H. Katz, Ion Stoica, and Amin Vahdat. FastLane: making short flows shorter with agile drop notification. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 84–96, 2015.

[72] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 17th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 78–85, 2017.

[73] Renjie Zhou, Dezun Dong, Shan Huang, and Yang Bai. FastTune: Timely and Precise Congestion Control in Data Center Network. In *Proceedings of the 2021 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 238–245, 2021.

[74] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 523–536, 2015.



(a) WKb Balanced

(b) WKb Core

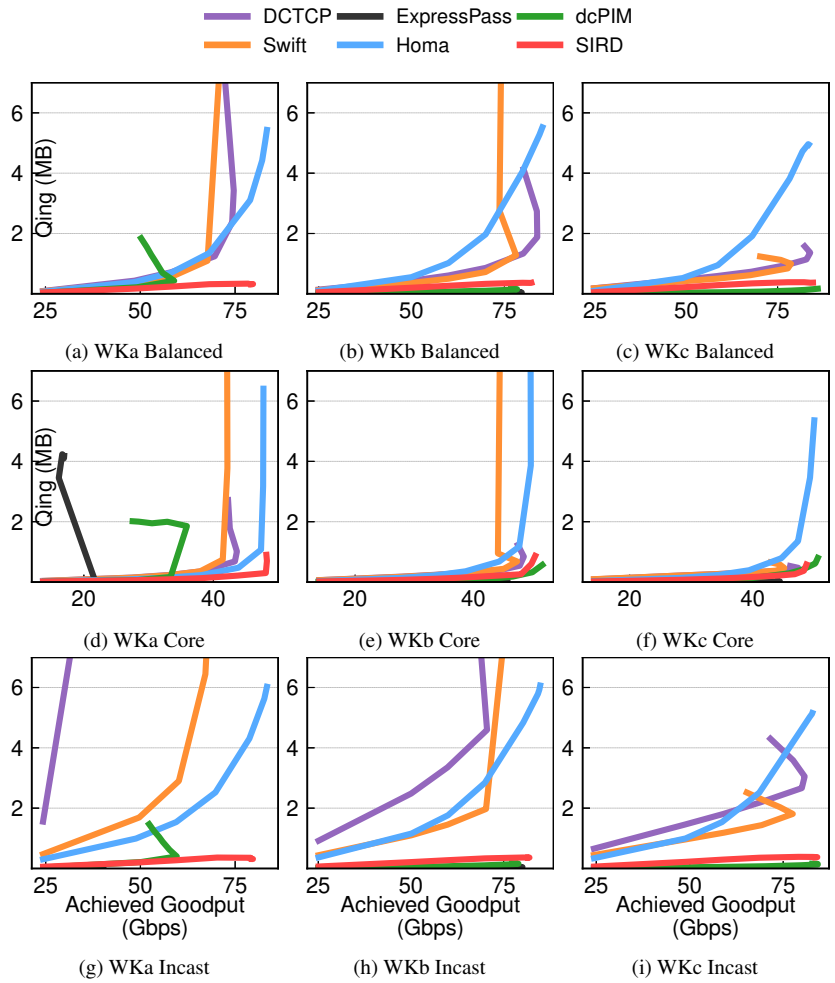
(c) WKb Incast

Figure 12: Median and 99^{th} percentile slowdown at 50% offered application load. Each bar group contains the messages in the following size ranges: $0 \leq A < MSS \leq B < 1 \times BDP \leq C < 8 \times BDP \leq D$.

A Appendix

Table 3: ASIC bisection bandwidth (in Tbps) and buffer sizes (in MB). Note that buffer architectures differ regarding the extent to which they are shared between ports.

	ASIC/Model	BW	Buffer
Broadcom	Trident+	0.64	9
	Trident2	1.28	12
	Trident2+	1.28	16
	Trident3-X4	1.7	32
	Trident3-X5	2	32
	Tomahawk	3.2	16
	Trident3-X7	3.2	32
	Tomahawk 2	6.4	42
	Tomahawk 3 BCM56983	6.4	32
	Tomahawk 3 BCM56984	6.4	64
	Tomahawk 3 BCM56982	8	64
	Tomahawk 3	12.8	64
	Trident4 BCM56880	12.8	132
	Tomahawk 4	25.6	113
nVidia	Spectrum SN2100	1.6	16
	Spectrum SN2410	2	16
	Spectrum SN2700	3.2	16
	Spectrum SN3420	2.4	42
	Spectrum SN3700	6.4	42
	Spectrum SN3700C	3.2	42
	Spectrum SN4600C	6.4	64
	Spectrum SN4410	8	64
	Spectrum SN4600	12.8	64
	Spectrum SN4700	12.8	64
	Spectrum SN5400	25.6	160
	Spectrum SN5600	51.2	160



(a) WKa Balanced (b) WKb Balanced (c) WKc Balanced

(d) WKa Core (e) WKb Core (f) WKc Core

(g) WKa Incast (h) WKb Incast (i) WKc Incast

Figure 13: Mean ToR queuing vs. achieved goodput. Configurations: Balanced (top), Core (middle), Incast (bottom).

Config	Default			Core			Incast			mean	range
Wload	WKa	WKb	WKc	WKa	WKb	WKc	WKa	WKb	WKc		
DCTCP	7.69	3.19	1.91	5.02	3.18	2.58	unstable	18.88	5.54	6.0	16.97
Swift	6.68	2.98	2.92	4.92	3.16	3.53	53.5	10.23	3.57	10.17	50.58
ExpressPass	unstable	14.94	8.09	unstable	15.9	11.37	unstable	12.61	8.2	11.85	7.81
Homa	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.42	1.27	1.08	0.42
dcPIM	1.67	2.98	1.84	1.58	3.45	2.68	1.66	2.05	1.98	2.21	1.87
SIRD	1.32	1.43	1.1	1.15	1.27	1.18	1.3	1.0	1.0	1.19	0.43
Normalized 99th percentile slowdown of all messages at 50% load.											
DCTCP	0.9	0.98	0.98	0.9	0.94	0.94	0.81	0.83	0.96	0.92	0.17
Swift	0.89	0.92	0.91	0.87	0.92	0.89	0.92	1.0	0.92	0.92	0.13
ExpressPass	0.46	0.93	0.93	0.45	0.89	0.88	0.48	0.94	0.95	0.77	0.5
Homa	1.0	1.0	0.97	0.99	0.96	0.99	1.0	0.99	0.98	0.99	0.04
dcPIM	0.71	0.92	1.0	0.74	1.0	1.0	0.71	0.92	1.0	0.89	0.29
SIRD	0.96	0.97	0.98	1.0	0.98	0.96	0.96	0.96	1.0	0.97	0.04
Normalized maximum goodput across applied load levels.											
DCTCP	unstable	85.81	9.71	3.68	62.13	7.94	unstable	unstable	260.51	71.63	256.83
Swift	unstable	unstable	8.41	unstable	unstable	10.12	unstable	unstable	125.77	48.1	117.36
ExpressPass	1.0	1.0	1.0	unstable	1.0	1.0	1.0	1.0	1.0	1.0	0.0
Homa	137.45	115.99	30.18	7.06	434.71	79.62	143.38	150.57	145.29	138.25	427.65
dcPIM	52.07	7.76	3.1	1.04	51.24	13.09	34.25	7.18	7.41	19.68	51.03
SIRD	12.05	9.91	2.68	1.0	38.9	7.53	9.37	10.03	10.27	11.3	37.9
Normalized maximum ToR queuing across applied load levels.											

Table 4: Normalized data used in Figure 5. Performance is normalized to the best performing protocol on each experiment. Experiments in which the protocol is unable to deliver the specified throughput or network queuing grows infinitely (unstable) are excluded from the calculation of mean and range.

Config	Default			Core			Incast			mean	range
Wload	WKa	WKb	WKc	WKa	WKb	WKc	WKa	WKb	WKc		
DCTCP	9.92	7.9	6.91	5.97	5.71	5.54	unstable	71.06	32.06	18.13	65.52
Swift	8.61	7.37	10.56	5.85	5.68	7.58	68.21	38.51	20.69	19.23	62.53
ExpressPass	unstable	36.94	29.26	unstable	28.56	24.42	unstable	47.45	47.45	35.68	23.03
Homa	1.29	2.47	3.62	1.19	1.8	2.15	1.27	5.36	7.37	2.95	6.18
dcPIM	2.16	7.37	6.66	1.88	6.2	5.76	2.11	7.7	11.48	5.7	9.6
SIRD	1.7	3.53	3.99	1.37	2.28	2.54	1.65	3.76	5.79	2.96	4.42
99th percentile slowdown of all messages at 50% load.											
DCTCP	74.65	83.85	83.95	43.63	48.54	47.42	67.8	70.53	80.74	66.79	40.32
Swift	74.52	78.16	78.69	42.16	47.41	45.15	76.8	85.46	77.57	67.32	43.3
ExpressPass	38.04	79.68	80.42	21.78	45.83	44.42	40.26	80.1	80.1	56.74	58.64
Homa	83.39	85.23	83.55	47.73	49.73	50.02	83.39	84.87	82.79	72.3	37.5
dcPIM	58.94	78.42	86.03	35.91	51.68	50.59	59.61	79.02	84.26	64.94	50.12
SIRD	79.74	82.27	84.71	48.27	50.47	48.75	79.7	81.98	84.13	71.11	36.44
Maximum goodput across applied load levels (Gbps).											
DCTCP	unstable	7.0	2.7	8.3	2.67	1.46	unstable	unstable	21.06	7.2	19.6
Swift	unstable	unstable	2.33	unstable	unstable	1.87	unstable	unstable	10.17	4.79	8.3
ExpressPass	0.06	0.08	0.28	unstable	0.04	0.18	0.08	0.08	0.08	0.11	0.24
Homa	8.63	9.46	8.37	15.91	18.68	14.68	12.17	12.17	11.75	12.42	10.31
dcPIM	3.27	0.63	0.86	2.34	2.2	2.41	2.91	0.58	0.6	1.76	2.69
SIRD	0.76	0.81	0.75	2.26	1.67	1.39	0.79	0.81	0.83	1.12	1.51
Maximum ToR queuing across applied load levels (MB).											

Table 5: Raw data used in Figure 5. Experiments in which the protocol is unable to deliver the specified throughput or network queuing grows infinitely (unstable) are excluded from the calculation of mean and range.