Dart: Divide and Specialize for Fast Response to Congestion in RDMA-Based Datacenter Networks

Jiachen Xue, Muhammad Usama Chaudhry, Balajee Vamanan, T. N. Vijaykumar, and Mithuna Thottethodi

***Abstract*— Though Remote Direct Memory Access (RDMA) promises to reduce datacenter network latencies significantly compared to TCP (e.g., 10x), end-to-end congestion control in the presence of incasts is a challenge. Targeting the full generality of the congestion problem, previous schemes rely on slow, iterative convergence to the appropriate sending rates (e.g., TIMELY takes 50 RTTs). Several papers have shown that even in oversubscribed datacenter networks most congestion occurs at the receiver. Accordingly, we propose a divide-and-specialize approach, called *Dart*, which isolates the common case of receiver congestion and further subdivides the remaining in-network congestion into the simpler spatially-localized and the harder spatially-dispersed cases. For receiver congestion, we propose *direct apportioning of sending rates (DASR)* in which a receiver for n senders directs each sender to cut its rate by a factor of n, converging in only one RTT. For the spatially-localized case, Dart provides fast (under one RTT) response by adding novel switch hardware for *in-order flow deflection (IOFD)* because RDMA disallows packet reordering on which previous load balancing schemes rely. For the uncommon spatially-dispersed case, Dart falls back to DCQCN. Small-scale testbed measurements and at-scale simulations, respectively, show that Dart achieves 60% (2.5x) and 79% (4.8x) lower 99^{th}-percentile latency, and similar and 58% higher throughput than InfiniBand, and TIMELY and DCQCN.**

***Index Terms*— Datacenters, RDMA, congestion control.**

I. Introduction

MANY modern, interactive datacenter applications have tight latency requirements due to stringent service-level agreements (e.g., under 200 ms for *Web Search*). TCP-based datacenter networks significantly lengthen the application latency. Remote Direct Memory Access (RDMA) substantially reduces latencies compared to TCP by bypassing the operating system via hardware support at the network interface (e.g., RDMA over InfiniBand and RDMA over Converged Ethernet (RoCE) can cut TCP's latency by 10x [1], [2]). As such, RDMA may soon replace TCP in datacenters [3]–[6].

Manuscript received November 27, 2018; revised June 14, 2019 and October 8, 2019; accepted December 10, 2019; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Tang. Date of publication January 14, 2020; date of current version February 14, 2020. *(Corresponding author: Balajee Vamanan.)*

Jiachen Xue was with the Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. He is now with NVIDIA Corporation, Santa Clara, CA 95051 USA (e-mail: xuejiachen@gmail.com).

Muhammad Usama Chaudhry was with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA. He is now with VMware Inc., Palo Alto, CA 94304 USA (e-mail: chaudhryusama@gmail.com).

Balajee Vamanan is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA (e-mail: bvamanan@uic.edu).

T. N. Vijaykumar and Mithuna Thottethodi are with the Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA (e-mail: vijay@ecn.purdue.edu; mithuna@purdue.edu).

Digital Object Identifier 10.1109/TNET.2019.2961671

Employing RDMA in datacenters, however, poses a challenge. RDMA provides hop-by-hop flow control and rate-based end-to-end congestion control [7], [8]. However, RDMA's congestion control is suboptimal for the well-known datacenter congestion problem, called *incast*, where multiple flows collide at a switch causing queuing delays and long latency tails [9] despite good network design [10], [11]. Though such congestion affects only a small fraction of the flows (e.g., 0.1%), datacenter applications' unique characteristics imply that the average latency is worsened. For example, because Web Search aggregates replies from thousands of nodes, the 99.9^{th} percentile reply latency affects the average response time; or alternatively, dropping the slowest replies worsens the response quality. In TCP, incasts cause delays due to packet drops and re-transmissions [9]. Though the lossless RDMA does not incur packet drops, incast-induced queuing delays lengthen RDMA's latency tail [12].

InfiniBand uses Early Congestion Notification (ECN) marks to infer imminent congestion and cuts back the sending rates [7], [8]. While DCQCN [12] proposes a similar scheme for RoCE, TIMELY [13] uses round-trip times (RTT) measurements, instead of ECN marks, for rate control in user-level TCP. Unfortunately, because ECN marks and RTT measurements need many round-trips to converge to the appropriate sending rates (e.g., 50 RTTs in TIMELY), the schemes are too slow for the applications' predominantly short flows each of which lasts only a handful of round-trips. During convergence, the schemes also lose throughput due to over- and under-shooting the sending rates.

To speed up convergence, we leverage the result in several papers [14]–[17] and reports from large datacenter operators such as Facebook [18], Google [19] and Microsoft [20]: even under typical oversubscription most congestion in datacenter networks occurs at the network edge (i.e., at the link from top-of-rack (ToR) switch to the receiver) as opposed to within the network. Our simulations confirm this result which is due to high-bandwidth network core [10], [11] and incast at the receiver. We make the key observation that while general congestion is complex and may require iterative convergence, the simpler and common case of receiver congestion can be addressed quicker via specialization; *Without isolating this case, previous schemes apply their iterative throttling to the general case. Instead, our proposal, called* Dart*, employs a divide-and-specialize approach to isolate receiver congestion and significantly speeds up the convergence.* Dart sub-divides the remaining case of in-network congestion into the simpler spatially-localized case and the harder spatially-dispersed case. For the former where the network capacity is not under pressure (e.g., due to imperfect ECMP hashing), Dart avoids

throttling which is unnecessary. For the latter where the network capacity is under pressure (e.g., due to dynamic network load spikes), Dart falls back on DCQCN's throttling which may be unavoidable. Load balancing [21]–[28] can alleviate localized in-network congestion but not receiver congestion, and usually reorders packets which is not supported by RDMA.

To address receiver congestion, we make the key observation that unlike in a wide-area setting, datacenter applications are co-operative where a receiver of n senders can direct each sender to cut its rate by a factor of n, This mechanism, called *direct apportioning of sending rates (DASR)*, ensures that the critical, short flows get their fair share of (instantaneous) throughput without being swamped by the background, long flows. When a sender completes, the (instantaneous) sending rate is adjusted as per the new sender count. Because DASR piggybacks the count in the receiver's acknowledgments to the senders, DASR achieves accurate and *one-RTT* convergence of sending rates without any repeated adjustments, unlike previous schemes. Specifically, (1) RCP [29] proposes to apportion the rates among the senders, but employs slow, iterative convergence *at the switches because RCP (a) targets general congestion without isolating receiver congestion and (b) uses general parameters to arrive indirectly at fair share instead of directly counting flows which is hard to do at Internet scales*; we evaluate RCP's convergence in Section VI-D. (2) EyeQ [15] highlights edge congestion but applies RCP's iterative convergence, which takes 25-30 RTTs, without specializing for edge congestion. (3) NUMFabric [30] achieves more flexible and faster bandwidth allocation than TCP but still employs iterative convergence (e.g., 31 RTTs). And, (4) while ExpressPass [31] and NDP [17] target general congestion via receiver-based congestion control, neither scheme isolates receiver congestion. ExpressPass employs BIC-TCP iterative convergence which takes 20 RTTs for a datacenter network (Section VI-D); ExpressPass shows results only for a simple network. NDP fundamentally relies on (a) packet spraying, which reorders packets, to reduce congestion and (b) packet trimming, which removes payloads, to unclog congestion notification to the receiver. Neither of these mechanisms is supported by RDMA which has no software stack like TCP. Without these mechanisms, NDP would see more congestion and slower feedback. DASR's faster convergence reduces latency tail (critical flows quickly get their share) and improves throughput (fewer adjustments). In an additional optimization, DASR leverages application-provided incast degree to avoid counting the senders and converge even faster.

To address spatially-localized, in-network congestion, Dart simply deflects the affected packets under the premise that an alternate path is faster than being queued up in the shortest path. To avoid livelock, Dart allows only a few deflections for a packet after which the packet is not deflected even at a congested switch. Dart avoids deadlocks via a widely-used virtual-channel-based scheme [8], [32]. Because RDMA does not support packet reordering, Dart provides hardware support in the switch to keep a flow's packets in order. While deflection [33] is well known, our contribution is *in-order flow deflection (IOFD)* unlike previous load-balancing schemes including DIBS [28]. As a congestion response, deflection is much lighter-weight and quicker (well under one RTT) than rate-cutting using iterative convergence and does not affect the sending rates. For spatially-dispersed in-network congestion, which is uncommon, Dart falls back to DCQCN's heavy-weight rate modulation. By filtering out receiver congestion and localized in-network congestion, Dart cuts the number of ECN marks, which trigger DCQCN fall-backs, by 4x for typical workloads.

We make four observations: First, receiver congestion is easy to differentiate from in-network congestion (Section III-C). Second, DASR works only for receiver congestion but not for in-network congestion (e.g., two flows collide in the network but go to different receivers which cannot detect the collision); and vice versa for IOFD (flows colliding at the receiver should not be deflected). As such, one of our contributions is identifying the specific case and applying the appropriate specialization. Third, because DASR and IOFD *separately* target receiver congestion and localized in-network congestion, respectively, they are more effective despite being simpler than previous schemes which tackle the full generality of the problem using a common mechanism. Finally, Dart leverages RDMA's unique features. While DASR is applicable to both RDMA and TCP, our DASR implementation relies on RDMA's discrete messages as opposed to TCP's continuous flows (Section III-B). IOFD specifically addresses RDMA's lack of support for packet reordering,

In summary, our key contributions are:

- employing a divide-and-specialize approach to congestion control;
- addressing receiver congestion via *direct apportioning of sending rates* by using the sender count to achieve accurate and faster, one-RTT convergence of sending rates than previous schemes which are iterative; and
- addressing spatially-localized in-network congestion via *in-order flow deflection* whereas previous schemes reorder packets which is not supported by RDMA.

A small-scale 16-node testbed implementation shows that Dart converges to the desired sending rate in one RTT and achieves 60% (2.5x) lower latency than and similar throughput as InfiniBand. Datacenter-scale *ns-3* simulations show that Dart achieves 79% (4.8x) lower 99^{th}-percentile latency and 58% higher throughput, on average, than TIMELY and DCQCN for typical over-subscription and load settings.

II. Challenges and Opportunities

Modern datacenter applications demand *both* low latency tails and high throughput from the network. Interactive datacenter applications, such as *Web Search*, generate thousands of short flows to lookup large distributed datasets for *each* user query. As described in Section I, the overall response time is bound by the 99^{th} - 99.9^{th} percentile of flow completion times (i.e., the tail-latency problem) [34]. Further, the synchronous nature of the lookup responses, which are aggregated in subsets, implies that each subset arrives at a switch causing an *incast*, which worsens when multiple queries' subsets arrive at the same time. On the other hand, background applications (e.g., Web Index update) demand high throughput for large volumes of Internet data. These long flows colliding with the short flows also exacerbate incasts.

The OS overheads in TCP drastically dilate network tail latencies (e.g., 99^{th} percentile latency is 10-20x of median latency [9]). Further, a slow response to congestion hurts latency at the start of incasts and throughput at the end. Similarly, an inaccurate response affects latency or throughput, depending on whether the rate was less or more than the optimum.

A. RDMA

With RDMA, the application invokes the NIC directly without involving the OS – (1) At the sender, the NIC uses DMA to copy data from the application memory to its buffers using DMA and sends the data after some protocol processing; (2) At the receiver, the NIC copies data into the receiving application's buffer. Thus, RDMA eliminates OS intervention and accelerates protocol processing at both the sender and the receiver. The buffers are pinned in physical memory and the address translations are cached at the NIC during connection establishment. RDMA-based transports [4], [5] show an order-of-magnitude reduction in flow latencies at low loads. As such, RDMA, initially proposed for multiprocessor networks [35], is finding its way into modern datacenters.

B. Challenges

Existing RDMA transports provide hop-by-hop flow control to ensure lossless operation. For example, InfiniBand [36] employs credit-based flow control and RoCE [37] uses Priority-based Flow Control (PFC). InfiniBand provides rate-based end-to-end congestion control using ECN marks [7], [8]. DCQCN [12] has shown that RoCE without end -to-end congestion control degrades in both latency and throughput at high loads.

As discussed in Section I, previous schemes address the full generality of the congestion problem and end up with iterative convergence to the appropriate sending rate upon congestion. Unlike TCP's window-based rate control, RCP's [29] routers iteratively calculate and convey the fair-share bandwidth to the senders sharing a link, which slows convergence (see Section VI-D). DCQCN [12] and TIMELY [13] improve end-to-end congestion control at datacenter scales for RDMA (RoCE) and user-level TCP respectively. Both DCQCN and TIMELY directly control the sending rate by pacing the packets sent out of the NIC. DCQCN starts a flow at the full line rate, employs ECN marks as feedback and cuts the sending rate in proportion to the exponentially-averaged fraction of ECN-marked packets. To avoid some problems of ECN (e.g., low-priority packets may not see ECN marks), TIMELY employs RTT measurements as feedback and modulates the sending rate (additive increase and multiplicative decrease) based on RTT gradients bounded by thresholds at the extremes.

Despite these innovative ideas, because these schemes tackle the general case with arbitrarily changing number of flows which interact in arbitrary ways, the schemes rely on slow, iterative convergence to the appropriate sending rates. As discussed in Section I, other schemes, including EyeQ [15], NumFabric [30] and ExpressPass [31], also rely on iterative convergence. Such convergence requires many round trips (e.g., 60 RTTs in RCP, 50 RTTs in TIMELY, 31 RTTs in NUMFabric, and 25-30 RTTs in EyeQ), as illustrated in Figure 1 for a sender whose initial sending rate is 100% of the line rate and the target rate is 50%. The upper half of Figure 1 shows the tuning of sender-inferred rates. Such iterative convergence hurts both latency and throughput, as we show in Section VI-B. Because DCQCN and TIMELY specifically target RDMA (RoCE) and user-level TCP (which bypasses the OS like RDMA), respectively, and are representative of iterative convergence, we compare Dart to these two schemes in our results.

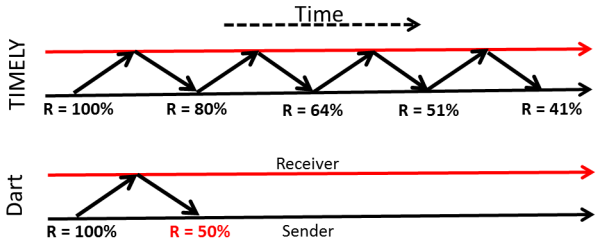

Fig. 1. Dart's fast, one-RTT convergence.

C. Opportunities

Dart employs a divide-and-specialize approach to avoid iterative convergence in the common case of receiver congestion (i.e., multiple senders intentionally sending to a receiver). For this case, Dart uses *direct apportioning of sending rates (DASR)* which specifies the appropriate sending rate in one RTT without repeated adjustments (see the lower half of Figure 1). Thus, Dart achieves accurate and fast convergence for receiver congestion. Dart further sub-divides the remaining case of in-network congestion into two sub-cases: the easier spatially-localized congestion and the harder spatially-dispersed congestion. For the localized sub-case, Dart employs in-order flow deflection (IOFD) which does not affect the sending rates. Such deflection is a quicker, lighter-weight, in-network response (well under one RTT) than the previous schemes' iterative convergence. For the dispersed sub-case, which is uncommon especially after IOFD filters out localized congestion, Dart falls back to DCQCN.

III. Receiver Congestion

We start with direct apportioning of sending rates (DASR) and describe in-order flow deflection (IOFD) in Section IV. We note that when contention is at the end-points, the fair share of bandwidth for each of n (say) senders is well-defined as $\frac{1}{n}$. The fair share can be extended easily to weighted fair shares.

In our description of DASR and IOFD, we use the term 'flows' to mean RDMA messages. Short flows are effectively small messages (e.g., those that contain small search queries for web-search, or key-value lookup requests for memcached). Long flows are effectively large messages that perform bulk-copying of large sections of memory (e.g., for index-updates in web-search). Both short and long flows may be packetized as necessary. While flow sizes are not known to the TCP layer, message sizes must be sent explicitly in RDMA and hence the RDMA application messaging layer can identify long and short flows.

A. Direct Apportioning of Sending Rates

All flows begin at the full line rate because (1) we want to avoid penalizing the latency of short flows, and (2) Dart's

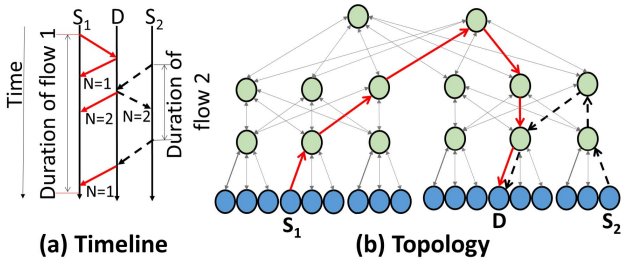

Fig. 2. Direct apportioning of sending rates.

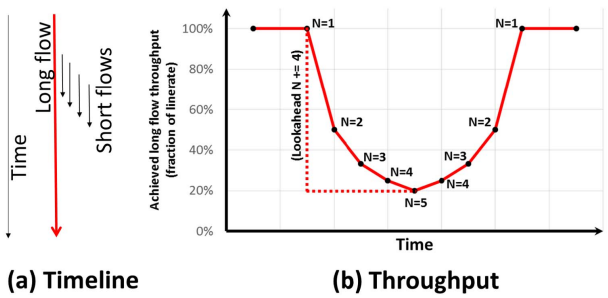

Fig. 3. Short flows mixed with long flows.

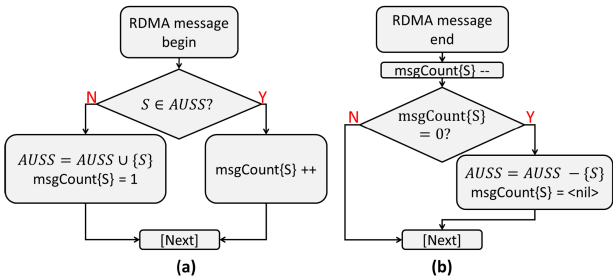

Fig. 4. Active Unique Sender Set for sender S (in software).

fast feedback can quickly throttle long flows if necessary. Dart piggybacks the sender count, the n value, with ACKs to all the senders; ACKs use high-priority queues in Dart as well as all the other schemes we compare. Such piggybacking can be achieved via NIC firmware without hardware changes. (In practice, implementing firmware changes on proprietary NICs is not feasible without vendor support. We discuss our prototyping approach later in Section V.) As such, senders receive continuous, fast – one-RTT – direction from the receivers on their allowed transmission rate. Such co-ordination is between the end-point NICs; the switches need not be modified.

Figure 2 shows an oversubscribed fat tree to illustrate Dart's operation in terms of fair-sharing among long flows. Consider the example shown in Figure 2(a) wherein a single receiver (D) receives a steady long flow from one sender (S_1) at the line rate. That sender continues to transmit at the line rate without throttling as it sees the n value remain 1 in the ACKs from the receiver. When a second sender (S_2) initiates another long flow to the same receiver (D), there is contention at the leaf-level switch, as shown in Figure 2(b) where the solid and broken lines show the two flows. As the two flows' packets arrive interleaved at the destination node, the receiver's NIC piggy-backs the updated $n = 2$ value with the ACKs to each sender. The ACKs cause the sender NICs to throttle the rate to $\frac{1}{n} = \frac{1}{2}$ of the line rate, which can be sustained in steady state.

The above discussion illustrates the two key benefits of DASR. First, the continuous feedback mechanism means that congestion control feedback to senders is fast, in one RTT. Second, the senders are given an accurate and precise rate not to exceed. The algorithm seamlessly handles flow "churn" by constantly sending updated n values.

B. Short Flows and Incasts Under DASR

The case of short flows, including incasts, interacting with long flows uses the same mechanism to ensure that the latency of short-flows is not hurt (Figure 3(a)). A long flow that contends with k other short flows from k unique senders is directed to reduce its sending rate to $\frac{1}{k+1}$ because $n = k + 1$. While this throttling helps the short flows' latency, such throttling is short-lived and does not hurt the long flow's throughput. The presence of short flows can be treated as a case of flow-churn; the long flows throttle their rates according to the number of short flows, but only for the duration of the short flows (Figure 3(b).

The rate throttling at the sender is staggered by the time required for the receiver's ACK (with the piggy-backed n value) to reach the sender. while DCQCN and TIMELY also incur this ACK delay (Section II), the previous schemes require several iterations of RTT measurements or ECN marks, involving several round trips, for the sender to infer the appropriate rate (e.g., 50 RTTs in TIMELY). This delay hurts both short flows' latency and long flows' throughput. In contrast, DASR converges in one RTT to the appropriate sending rates.

Dart addresses one other challenge: accurate counting of senders. Consider a case where two incasts to the same destination (say D) begin close in time and there is an overlap in the senders of the two incasts (sender S is part of both incast groups). Because S's two incast flows would be serialized at S's NIC, D's NIC should count source S exactly once when determining n. This case is handled naturally because Dart tracks *in software* the unique senders of active flows – in the Active Unique Sender Set (AUSS). Upon a new message/flow, the sender of the message is added to the AUSS if not already present (see Figure 4(a)). Further, Dart initializes a count of in-flight messages associated with that sender to 1 (if not previously present in the AUSS) or increment the in-flight message count (if previously present in the AUSS and multiple messages from the same sender are concurrently active). Dart finally decrements the sender count only when all the messages from that sender terminate, as shown in Figure 4(b). With the above tracking in place, DASR can use the number of elements in the AUSS as the n value (i.e., $n = |AUSS|$).

Finally, each sender in the AUSS is associated with a timestamp of the flow's last packet. Any flow that is idle for long (e.g., 2 seconds) is assumed to be dead and eliminated from the AUSS. This well-known soft-state approach ensures that DASR does not artificially throttle active senders in cases where other senders may fail after initiating message transmission. Recall that RCP requires switch support to handle the full generality. In contrast, Dart requires extra state only at the receiver (host) to specialize the common case of receiver congestion.

RDMA's connectionless nature (unlike TCP) and its clearly-marked message start/end ensures that senders are not counted

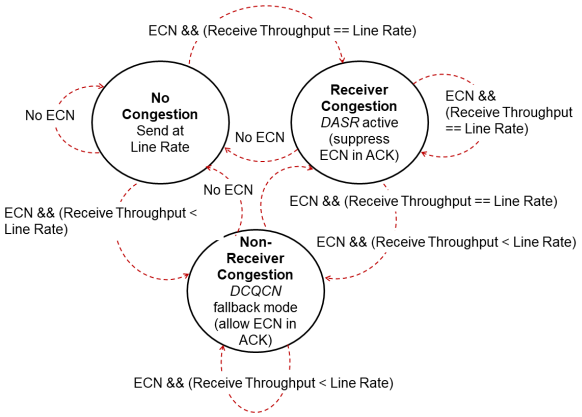

Fig. 5. Handling receiver and non-receiver congestion.

in idle periods (as shown in Figure 4). Because our DASR implementation relies on RDMA's message start/end markers for accurate AUSS tracking, it does not extend to TCP which views communication as a continuous stream without markers making it hard to account for flow idleness.

C. Handling Non-Receiver Congestion

Figure 5 illustrates our state machine that *exhaustively* handles receiver and non-receiver (in-network and at source) congestion. Dart distinguishes between receiver and non-receiver congestion based on two observable symptoms: (1) throughput at the receiver, and (2) ECN marks. Changes in either of the two trigger state changes as shown in Figure 5.

As long as no ECN marks are received, Dart remains in the "No Congestion" state. While DASR targets receiver congestion, both receiver congestion and non-receiver congestion (including network and source congestion at the sender's NIC) may result in ECN marks. For source congestion, we require that the source NICs be capable of ECN marking, which is possible in today's SmartNICs. For example, we can programmatically set ECN on Netronome Agilio CX NICs based on queue depth, which is accessible as intrinsic metadata [38]. Without any additional safeguards, the ECN-based DCQCN fall-back may over-throttle the sending rates in addition to DASR even for receiver congestion. To avoid such over-throttling, we observe that during receiver congestion, the throughput seen by the receiver is not affected as all flows headed to that receiver would be serialized anyway at the last hop (i.e., the receive throughput is equal to line rate). This condition triggers DASR, denoted by "Receiver Congestion" state in Figure 5. In this state, Dart piggy-backs the n values while suppressing the ECN marks on the returning `ACKs`. The throughput is unaffected even if receiver congestion occurs at an internal switch – it is still receiver congestion irrespective of where it occurs.

In contrast, in the case of non-receiver congestion (i.e., network and source congestion) where contending flows are headed to different destinations, the bottleneck link capacity would be shared by contending flows. As a result, when the flows eventually reach their destinations, the receivers would observe throughputs that are less than the line rate. In addition, the receiver would also observe ECN marks due to congestion. Accordingly, Dart enters "Non Receiver Congestion" state when the receiver observes lower than line rate as well as ECN marks. Because DASR cannot handle non-receiver congestion, the receiver allows ECN marks, which trigger DCQCN at the senders. Finally, to avoid DASR from interfering with DCQCN, Dart sets $n = 1$ in this state. While the above description handles receiver and non-receiver congestion occurring separately, Dart naturally handles the case of the two together in two steps. In the first step, Dart enters the "Receiver Congestion" state causing DASR to kick in. For non-receiver congestion, however, DASR's apportioning may cause the senders to underutilize their throughput share. In that case, the receiver rate would fall below the line rate, causing a transition to the "Non-Receiver Congestion" state in the second step, where DSQCN kicks in to avoid continued throughput loss. Thus, Dart *exhaustively* covers all cases of congestion among the three states in Figure 5.

Dart's convergence: From Figure 5, it is clear that DASR covers only the special case of receiver congestion and converges to the *correct* sender rate (i.e., fair share). During non-receiver congestion (i.e., in-network or source congestion), Dart falls back to DCQCN. Dart's convergence is thus guaranteed by DCQCN's convergence in this case. Overall, because our state machine exhaustively covers *all* congestion states, Dart converges to the correct sender rates in *all* cases.

D. Accelerated DASR

We further improve Dart's performance by having the application provide a *look-ahead* notification of the upcoming set of incast flows that are part of an incast group. For example, if each incast message carries (1) information that it is part of a 20-flow incast and (2) the list of the 20 senders, the receiver NIC can advertise rate limits to the 20 senders after just the first such message, even before the other senders' packets arrive at the receiver. As with the n value, such lookahead notification can also be handled via NIC firmware. Thus, the AUSS can be populated with the set of senders in advance of actual packet arrival from all the senders. The long flows back off quicker with this look-ahead, as shown by the dotted line in Figure 3(b). For accurate counting, Dart treats any flow as if it begins when the look-ahead notification first arrives. The ending of flows is handled as without the look-ahead. The look-ahead overhead is reasonable (e.g., 20 two-byte sender-ids, each of which can address 64K sender NICs, amount to 40-byte or 2% overhead for a 2-KB payload). Unlike generic applications, latency-sensitive applications are specialized where the incast groups – *static* in the application – are likely known to the programmer (e.g., Web Search). Identifying the static groups is enough even if they *dynamically* and unpredictably break into subsets at different switches because eventually the whole static group causes receiver congestion which is DASR's target.

E. Failures and Attacks Under DASR

Because the AUSS tracking uses soft-state (as described in Section III-B), Dart can handle failures seamlessly. Any flow in the AUSS (irrespective of whether it uses look-ahead) will naturally timeout and exit the AUSS when senders fail. However, untrusted entities in multi-tenant datacenters may

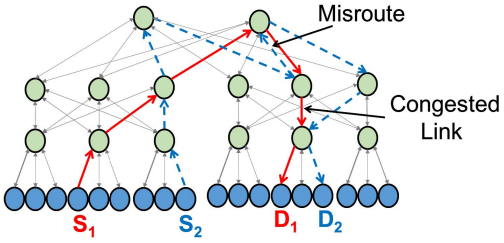

Fig. 6. Misrouting to avoid congested links.

attempt denial-of-service attacks by frequently sending look-ahead notifications which results in other senders throttling themselves. To ensure SLA compliance, datacenters typically use rate-limiting to ensure that VMs of a tenant do not exceed their fair share of bandwidth. Dart's AUSS tracking can be private to individual tenant's flows. As such, any false information from one tenant can not affect other tenants' flows. As a last resort, the look-ahead optimization can be turned off in multi-tenant datacenters, while retaining the main DASR which is not susceptible to such attacks. We isolate the look-ahead's performance from that of the main DASR in Section VI-C.

IV. Localized In-Network Congestion

We now address in-network congestion, starting with the easier spatially-localized congestion, including incasts, and then discuss the harder spatially-dispersed congestion. Localized in-network contention is usually the result of temporary link contention in a small neighborhood of switches. Such contention may result in packets being unnecessarily serialized (e.g., even though they may be headed to different destinations). In such situations, Dart deflects *all* the packets of selected short-flows to avoid this serialization penalty. Consider the example shown in Figure 6 with two flows between the source-destination pairs (S_1, D_1) (solid arrows) and (S_2, D_2) (dashed arrows). Assuming the second flow (dashed arrows) finds one of the links congested, the flow may take an alternate path, away from the congested link – a response well under one RTT. While such deflection results in additional hops (two in the example – one misroute and another to recover from the misroute), Dart's deflection policies ensure that (1) this penalty is far lower than that of the serialization so that deflection significantly improves latency over previous schemes' iterative convergence, and (2) the relative overhead of extra link utilization is low (Section IV-B). Further, our design is free from livelocks and deadlocks (Section IV-B). We describe below Dart's mechanisms and policies for such deflection-based congestion avoidance.

A. In-Order Flow Deflection Mechanisms

Deflection routing is a well-known technique for load balancing [33]. In general, deflection routing can cause reordering of packets. As such, deflection is relatively straightforward to use when either the application does not require ordered packet delivery or there is a reassembly layer that reorders received packets to be delivered in the correct order (e.g., TCP). Indeed, in addition to being well-explored in other contexts, such packet-by-packet deflection has also been proposed for congestion avoidance in data centers (DIBS [28]).

In contrast, for RDMA networks, there is no software stack to reassemble out-of-order packets of a message/flow. Consequently, the limited hardware support for packet reordering are cases where re-ordering does not change the semantics. For example, current support in ConnectX5 [39] is limited to read/write RDMA verbs. A bulk remote-write can be broken into many pieces and the order among the pieces is not important as long as they all complete. However, such reordering can change the semantics in send/recv based RDMA verbs. For example, a flow abstraction, if broken up into pieces, needs (1) sequence numbers associated with each piece, and (2) reassembly at the receiver to put the pieces back together in sequence order. Note that send/recv verbs are widely used *and* acknowledged as higher performing than read/write for server applications [5]. Recent work by Mittal *et al* [40] extends modest support even for reordering of flows with such sequence numbers; but with a fixed hardware window for reassembly. Such limited window size requires the source to throttle packets to ensure that the sliding window does not overflow, which reduces throughput. On the other hand, packet-by-packet deflection would require the ability to handle unbounded reordering (and not just the limited reordering support in [40]), which can impose significant CPU overheads [26].

To avoid such overheads, the network must guarantee in-order delivery semantics. For such networks, Dart uses novel *in-order flow deflection (IOFD)* instead of the above packet-level deflection. The key challenge in IOFD is to ensure that later packets of the flow traverse the same network path as the header packet of the flow. Further, the semantics do not allow for any false-positives (i.e., the switch misidentifies a non-deflected flow as a deflected flow) or false negatives (i.e., the switch 'forgets' a misrouted flow to be one). Such strong semantics may seem challenging especially when considering router failures. We describe the fault-free case below and address faults in Section IV-C.

A naive solution would be to maintain routing history in the switch for every flow which may be many at a given time, and look up the history for every packet. Fortunately, because only short flows are latency-critical, IOFD applies only to short flows only a few of which overlap at a switch at any given time (say 4 to 8). Long flows that collide at the receiver are handled by DASR. Some spatially-dispersed in-network congestion due to long flows is inevitable despite best-in-class hashing and other schemes [21], [25], [26]. In our design, such collisions trigger the DCQCN fall-back. Crucially, the latency-critical short flows are deflected away from such collisions. Recall that flow sizes are known in RDMA (Section III).

IOFD maintains the set of misrouted flows in a small content-addressable memory (CAM) called the *deflected flow table (DFT)* at each router. Entries in the DFT are allocated when the start packet of an RDMA message is chosen for deflection *and* a free entry is available in the DFT. Each entry includes the flow id or RDMA message id (the searchable field) and a randomly-selected output port for that flow (the data field of the table entry). Entries in the DFT are de-allocated when the end packet of an RDMA message passes through the switch. To ensure that the history of misrouted flows is not lost, no DFT entry may be overwritten

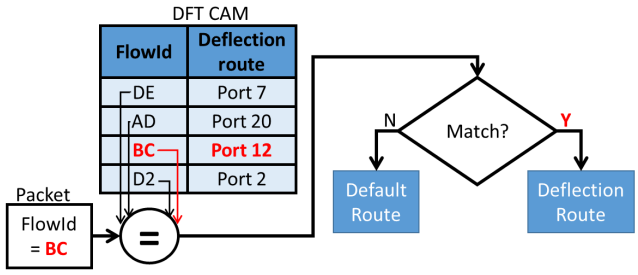

Fig. 7. Packet routing with DFT lookup (in hardware).

except by natural deallocation. To avoid livelocks, IOFD allows a packet only a limited number of *misroutes* which are encoded as *deflection token* bits in each packet header. The switch removes a token from each misrouted packet. If either DFT entries are unavailable or the flow has exhausted its tokens, the flow may not be deflected at the switch. Every packet consults the DFT to determine its path, as shown in Figure 7. If the packet's flow-identifier matches one of the entries, the packet is deflected to the port indicated by the entry (e.g., a packet matching DFT entry id = 0xBC is deflected to port 12 in Figure 7). To ensure that messages do not end up at unintended end-nodes, leaf-level switches (ToR switches) deflect messages back to the network and not to end-nodes. Note that, because the DFT is small (e.g., 8-entry CAM), the delay and power overheads are negligible.

B. IOFD Policies

There are three policy decisions that IOFD makes to strike a balance between over-aggressive deflection and inadequate deflection. First, to determine if deflection is competitive (i.e., the expected queuing delay at the switch is high enough that a few additional network hops may be better), IOFD compares the current queue position to an empirically-determined *deflection threshold*. Deflection is allowed only if the queue position is above the threshold. Second, to avoid unnecessarily-long deflection chains and livelock caused by loops, IOFD deflects only the packets with spare deflection tokens (Section IV-A). Once the deflection tokens are exhausted, a packet incurs the full latency penalty of waiting in the network queues. Finally, the possibility of deadlocks must be carefully handled. Specifically, modern DC networks (Clos variants [41]) typically use Valley-free routing [42] or up*/down* routing [43] to guarantee deadlock-freedom. Although IOFD can violate the rules of valley-free-routing, such violations are possible independent of IOFD. Hu *et al* [44] show the violations of valley-free routing in real data-center measurements for a RoCE network. As such, IOFD can leverage the same (or similar) mechanisms that are used to handle failures to handle flow deflections. We employ one such well-understood *deadlock avoidance* [32] technique by leveraging virtual channels ('virtual lanes' (VL) in InfiniBand [8]). Deadlock avoidance employs two class of VLs: (1) escape VLs that are guaranteed to avoid cyclic buffer dependencies and (2) non-escape VLs that may incur cyclic buffer dependencies. Packets/flows may move from non-escape VLs to escape VLs (which ensures that flows in non-escape VLs can always make forward progress) but *not* vice versa. In our context, if traffic on one virtual lane (VL) – the escape VL – is not deflected, and flows that traverse the escape VLs never flow back to non-escape VLs, deadlock-freedom is guaranteed. Unlike *deadlock prevention* which places routing restrictions that avoid certain turns (e.g., [42], [45]), deadlock avoidance works without preventing any turns [32], [43]; rather it takes the approach that any turn may be allowed by at least some VLs. Recent work [46] discusses deadlocks, *other* than routing deadlocks, created by extraneous reasons such as SDN updates, BGP re-routes, and misconfigurations. Such deadlocks can occur despite deadlock-free routing and must be solved separately (e.g., via sound SDN updates).

IOFD does not misroute long flows. Misrouting is a latency optimization for short flows only. Unlike short flows, long flows are sensitive to throughput not latency. Also, long flows are a dominant fraction of network load, and, therefore, deflecting long flows to longer paths would overload the network. We achieve this restriction by setting the number of deflection tokens to zero for long-flow packets. Deflecting only short flows only a few times ensures that the increase in link utilization and path dilation due to IOFD are modest, as shown in Section VI-B.

Finally, if IOFD succeeds in dissipating localized congestion then DCQCN does not kick in (i.e., no ECN marks). Otherwise (e.g., deflection tokens exhausted), the flows incur ECN marks which trigger the DCQCN fall-back. To ensure that IOFD is activated before ECN marks are triggered, IOFD's deflection threshold is lower than the ECN threshold. Our results in Section VI-B show that Dart (DASR and IOFD) cuts the number of ECN marks, which trigger DCQCN fall-backs, by 4x (i.e., the fall-back is infrequent; otherwise, Dart would not perform better than DCQCN).

C. Failures Under IOFD

Because each deflected flow's meta-state is distributed across multiple routers' DFTs, router failures must be correctly handled. To understand how IOFD handles router failures, let us consider how conventional RDMA handles failures. The back pressure of InfiniBand/RDMA networks ensures that packets queue up at upstream routers (and do not get dropped). The neighboring routers detect a failed router and propagate that information back to senders and effectively cause the in-flight packets to be dropped. For reliable (i.e., RC) communication, the senders must re-transmit the messages whose completion events have not been received). This approach carries over to IOFD without changes irrespective of whether flows have been deflected. As in the baseline case, flows blocked by failures are not allowed to locally reroute around the failed routers (which could cause ordering violations). Instead, all such blocked flows are effectively dropped and must be re-transmitted by the senders.

V. Small-Scale Measurements

Dart has two key components: DASR which does *not* need any hardware switch changes and IOFD which does. Accordingly, we implement DASR in our small testbed as we lack access to datacenter-scale networks (this section). Because hardware changes are hard to implement for a paper, we simulate IOFD, and the full Dart, at datacenter scales using ns-3 (Section VI-A).

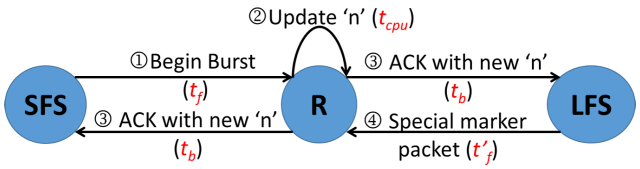

Fig. 8. DASR convergence time measurement.

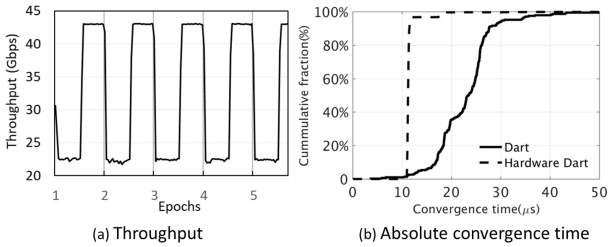

Fig. 9. Testbed measurements of DASR convergence.

Our testbed consists of 20 nodes, each consisting of four eight-core AMD Opteron 6320 CPUs running at 2.8 GHz and 256 GB of memory, which connect to a 36-port *Mellanox SX6025 InfiniBand* switch using Mellanox ConnectX-3 Pro HCA. The switch provides bidirectional bandwidth of 56 Gbps per port. All the nodes run RHEL6.7 (kernel version 2.6.32) and Mellanox OFED 3.3-1.0.4.

We conduct one experiment to evaluate DASR's convergence (Section V-A) and another to evaluate DASR's performance in the presence of incast (Section V-B). Implementing DASR in our testbed via firmware changes on proprietary NICs is infeasible without vendor support. Fortunately, because we do not have multiple applications in our testbed, we implement AUSS in the application layer, in which senders and receivers exchange n values using application-level acknowledgements. Further, we evaluate accelerated DASR only using simulations, and not in our testbed.

A. DASR's Convergence

We answer two key questions: (1) whether DASR converges to fair share bandwidth, and (2) whether it converges fast. We use two senders (Figure 8) – a long-flow sender (LFS) and a short-flow sender (SFS) – and a receiver (R). While LFS continuously sends to R, SFS starts a new transmission, taking t_f to reach R, which then takes t_{cpu} to recalculate the new n value. Finally, the updated n value is received at both SFS and LFS, which then adjust their sending rates, all of which takes t_b. The convergence time is the sum of t_f, t_{cpu}, and t_b. However, because the key events occur at different servers with independent clocks, the time components cannot be determined accurately from the events. Therefore, we map the multi-server events into meaningful single-server measurements at R. First, instead of measuring t_f, we measure t'_f for a specially-marked message from LFS to R indicating that LFS has seen the new n value. t_f and t'_f are equal because SFS and LFS are equidistant from R and those paths are not congested (if anything, LFS to R may be loaded more than R to SFS so that $t'_f > t_f$ making our measurements conservative). Second, upon receiving SFS's first message at R, we measure t_{cpu}, t_b, and t'_f, which also add up to the convergence time.

LFS constantly sends 64-KB messages to R. Later, SFS sends periodic bursts, during which both SFS and LFS drop to 50% of the line rate. Each burst consists of 32K messages of 64 KB each. We define the time to send such a burst as an *epoch*. We measure throughput for groups of 1K messages because per-message bandwidth measurement is extremely noisy. SFS, LFS and R run on separate nodes.

Figure 9(a) plots LFS's throughput (Y-axis) over time in epochs on the X-axis. The vertical grid lines correspond to SFS's bursts. In the absence of contention, LFS achieves 43 Gbps which is the peak throughput achieved in our testbed for our message/batch size. However, when SFS sends its periodic traffic, LFS near-instantaneously throttles itself to approximately half the sending rate (22.5 Gbps). As soon as SFS stops, LFS goes back to the maximum rate. We measured 1K bursts from SFS (which are seen as troughs in LFS's throughput) but show only five to avoid clutter.

Figure 9(a) is not a good indicator of the absolute convergence time because the throughput is averaged over groups of 1K messages. As such, we directly measure DASR's absolute convergence times in each of the 1024 epochs. Figure 9(b) shows the distribution (solid line) of our 1K measurements of the convergence times (in μs on X-axis). The 90^{th}, 99^{th}, and 99.9^{th} percentile convergence times are 28 μs, 41 μs, and 44 μs, respectively. The unloaded RTT is 15 μs. In contrast to DASR's one-RTT convergence, TIMELY's convergence takes 50 1-ms RTTs. Figures 18 and 2 in TIMELY [13] show 50-ms convergence and the worst-case RTT to be 1 ms, respectively. Similarly, RCP [29] and ExpressPass [31] require several RTTs to converge; we study their convergence in Section VI-D.

The above convergence time is for our DASR implementation which maintains the AFS in software (Section III-B). We also show a dashed line in Figure 9(b) which depicts the convergence time for a NIC hardware implementation of DASR. Here, RDMA's built-in completion queues notify a sender that communication is complete which is faster than in software. Then, our convergence time would approach the hardware-RTT (12 μs).

B. DASR's Incast Performance

We compare the completion times of short, incast flows and throughput of long, background flows of InfiniBand and DASR. We initiate short 256-KB incasts from a group of servers every 100 ms to an *aggregator* server. Meanwhile, we send continuous background traffic from another server to the aggregator. We introduce random jitter of 0-100 μs among the incast senders in each round. While InfiniBand uses its congestion control [8], we implement DASR's rate control by staggering the messages in time at the application layer. Here, we do not compare to DCQCN or TIMELY which require NIC firmware changes and special timer hardware, respectively; we simulate them in Section VI-A.

Figure 10 shows the median and tail (99^{th} percentile) flow completion times of DASR and InfiniBand (Y-axis), for varying incast degrees (X-axis). As expected, higher incast degrees lead to longer flow completion times and even longer tails. DASR reduces the medians and tails by 2.5 - 3.3x. DASR's reductions in the tails are close to those in the medians because the tails are only about 1.2x longer than the medians

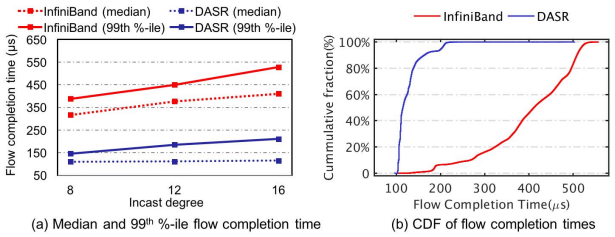

Fig. 10. Testbed flow completion latency.

in InfiniBand due to our testbed's (small) scale. As the tails grow at datacenter scales (e.g., 5-10x of the median), DASR achieves greater tail reductions (e.g., 5x in Section VI-B). Figure 10(b) shows the flow completion time distributions of InfiniBand and DASR for the incast degree of 16. As compared to InfiniBand, DASR reduces the spread and shifts the curve to the left. Both DASR and InfiniBand achieve similar throughput (within 0.5%) for long flows (not shown).

VI. At-Scale Simulations

We evaluate *Dart* , DCTCP (includes OS overheads), DCQCN, and TIMELY using typical datacenter traffic patterns [14].

A. Simulation Methodology

Simulated Network: We simulate a datacenter with 1024 hosts that are connected in an over-subscribed Clos topology [41]. As per common practice, we use (1) an over-subscription factor of 4 [10], (2) $10\,Gbps$ point-to-point links with a propagation delay of $5\ \mu s$ so that the longest path is 6 hops or $30\ \mu s$, and (3) shallow, $225\ KB$ switch buffers and accordingly the ECN threshold of $22.5\ KB$ (i.e., 10% of the buffer size) [47], [48]. To utilize all the fat tree paths, we enable Equal Cost Multi-Path (ECMP) routing. Dart adaptively deflects packets, in addition to ECMP.

Workload: We model our workloads based on real datacenter production traffic characteristics [14] and similar to TIMELY's [13]. Section 4 in [14] lists MapReduce and Web applications as the applications that create the traffic. Specifically, we follow both the flow size distribution as well as the background/foreground traffic mix from [14]. To model background traffic (e.g., Web Index update), each server initiates a long flow of size $1\ GB$ with a randomly-chosen receiver. Our foreground traffic that models interactive applications uses short flows of size uniformly chosen among $\{2\,KB, 4\,KB,$ and $8\,KB\}$ with a default incast-degree of 16 (varied later). Further, groups of randomly-chosen servers send to randomly-chosen receivers causing multiple incasts which are typical (e.g., in Web Search). Further, we vary both the overall network load and the split between background (long) and foreground (short, incast) flows.

DCTCP: Our DCTCP implementation is built over TCP New-Reno. We set the initial congestion window to be 10 segments and the re-transmit timeout to $10ms$ (typical). We model an OS overhead of $300\ \mu s$ for each data transfer and calibrate our DCTCP latencies to match those reported by DCQCN.

TIMELY: We implemented TIMELY on *ns-3* where the RTT measurements are precise (i.e., we avoid the measurement issues discussed in the TIMELY paper). While TIMELY uses 64-KB segments to amortize the cost of NIC offload which is not modeled in *ns-3*, we use smaller 1460-byte segments which provides finer rate control and only improves TIMELY's performance in our runs. To reduce implementation complexity, we use a window-based implementation which sets the window size based on TIMELY's desired sending rate. We set TIMELY's parameters as per the TIMELY paper: $T_{low} = 50\ \mu s$, $T_{high} = 500\ \mu s$, $\alpha = 1\ Mbps$, and $\beta = 0.8$. We also modeled Hyperactive Increment (HAI) for flows to quickly ramp-up their rates.

DCQCN: DCQCN utilizes ECN to infer congestion, similar to DCTCP but with different thresholds. On receiving ECN, our simulated receivers run the Notification Point (NP) algorithm and generate Congestion Notification Packets (CNP) back to the sender if needed using high-priority queues. The receivers generate at most one CNP packet every $50\mu s$, as specified by DCQCN. On receiving a CNP packet, the senders calculate their target rate based on DCQCN's Reaction Point (RP) algorithm. Following DCQCN's recommendations, we set the exponential averaging factor, g, to $1/256$, the *byte counter* and *Timer* to be $10\,MB$ and $55\,\mu s$, respectively. Flows start at the line rate (i.e., there is no slow start). Finally, similar to TIMELY's HAI, there is a hyper-increase phase to quickly ramp-up the sending rates.

Dart : Dart leverages DASR and starts flows at the full line rate (Section III). We use an 8-entry deflected flow table (DFT); because we enable IOFD only for short flows (i.e., 2 – 8 KB flows), only a few misrouted flows co-exist at a switch (Section IV-A). To ensure that the light-weight IOFD occurs before the ECN-based heavy-weight response (Section IV-B), we set the deflection threshold to be 15 KB (ECN threshold is 22.5 KB). Because we experimentally found that our IOFD's benefits diminish after four misroutes, we set the deflection token count to be 4 (Section IV-A).

To avoid congestion in the reverse (i.e., `ACK`) path for ECN marks in DCTCP and DCQCN, RTT measurements in TIMELY, and n values in Dart, we use high-priority queues only for `ACKs`, as suggested by TIMELY.

B. Latency and Throughput

Figure 11 plots the 99^{th} percentile flow completion latency (Y-axis) for all the schemes (individual curves) under various load mixes using 8-KB short flows (the three sub graphs) and load levels (X-axis). We show the 8-KB flows out of the mix of 2-, 4-. and 8-KB flows as described in Section VI-A; we cover the others in Section VI-C. Note that the scales of both axes are different for the subgraphs because the network saturates differently across load levels. Figure 12 is similar to Figure 11 but it shows the median latency on the Y-axis.

Latency: For the typical load-mix (40% short flows, 60% long flows), as shown in Figure 11(a), Dart consistently achieves the lowest tail latency at the pre-saturation loads of 20% and 40% with a mean reduction in tail latency of 82% (5.6x); the range varies from 79% – 89% reduction over all the other schemes. Dart's (mean) reduction in tail latency is 79% (4.8x) when compared with DCQCN and TIMELY (i.e., ignoring DCTCP). Dart's DASR avoids iterative convergence for receiver congestion to arrive accurately and quickly – in one RTT – at the appropriate sending rate.

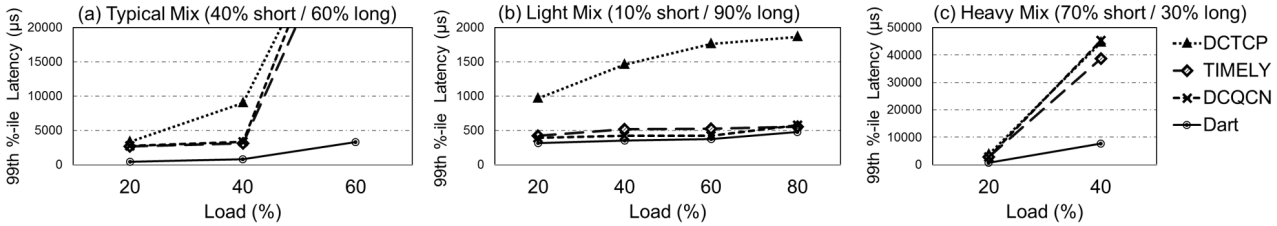

Fig. 11. 99^{th} percentile flow completion latency.

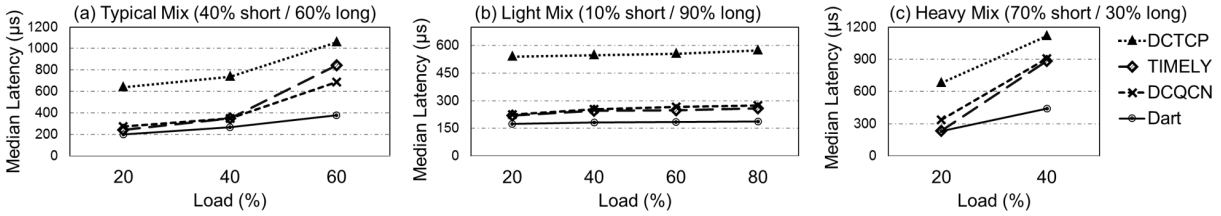

Fig. 12. Median flow completion latency.

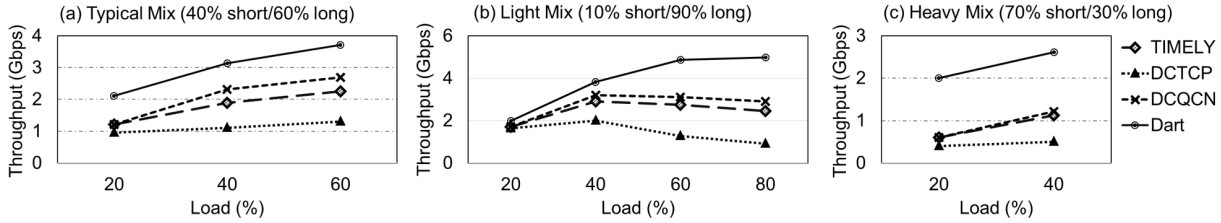

Fig. 13. Throughput.

We found that 72% of ECN marks in DCTCP occur at the ToR-receiver links confirming the key result that receiver congestion is the common case [14]–[20]. Further, Dart's IOFD provides quick response to avoid spatially-localized in-network congestion. Thus, Dart's divide-and-specialize approach using these two techniques achieves lower latency than TIMELY and DCQCN. Further, Dart delays the point of saturation past 60% load where DCQCN, TIMELY, and DCTCP saturate. DCQCN and TIMELY are similar because both rely on iterative convergence of the sending rates differing only in the congestion signals (ECN marks versus RTT measurements as mentioned in Section II-B); their median latencies and throughputs differ more (analyzed later). As expected, both are better than DCTCP, which incurs high operating system (OS) overhead avoided by the other schemes.

Figure 11 (b) and (c) illustrate the behavior when the load mix is made lighter or heavier, respectively, in terms of short flows (incasts). For the light load mix (Figure 11(b)), DCQCN, TIMELY, and Dart perform comparably because there is not much room for improvement. Due to its high OS overhead, DCTCP's latency penalty remains qualitatively similar to that for the typical load mix. For the heavy load mix (Figure 11(c)), Dart achieves 77% to 83% lower tail latency than the previous schemes. Further, while the previous schemes saturate above 20% load, Dart's latency increase is more modest as Dart extends the point of saturation.

One trend across the load mixes is that the network saturates earlier at higher short-flow fractions. This trend is not surprising as short flows do not offer sufficient time to take reactive action. (On the other hand, proactive methods such as slow-start would introduce unnecessary latency for short flows.)

Dart achieves significantly lower median latency at all load levels and load mixes as well (Figure 12). On average, Dart achieves 30% (1.4x) lower latency than DCQCN and TIMELY and 66% (3x) lower latency than DCTCP for the typical load mix. For the light mix and the heavy mix (Figure 12(b) and Figure 12(c), respectively), the latency reductions are 36% and 29%, respectively. Dart's improvements in median and tail latencies are higher here than in our testbed experiments (Section V-B) primarily because the larger scale provides more opportunity. DCQCN and TIMELY differ modestly in the median latencies in some cases. Median latency reduction indicates throughput improvements, as we see next.

Throughput: Figure 13 shows the throughput achieved for the same set of load levels and load-mix ratios. Figure 13(a) shows that Dart consistently outperforms the DCQCN and TIMELY. The mean improvement in throughput is 48% and 68%, (mean across all load levels) over DCQCN and TIMELY, respectively. DASR's accurate and one-RTT convergence is the key reason for Dart's higher throughput. IOFD directly improves only the latency and affects the throughput only indirectly by avoiding DCQCN fall-back which would cut the sending rates. As with latency, DCQCN and TIMELY outperform DCTCP in throughput due to DCTCP's OS overheads. With the heavy mix (Figure 13(c)), Dart is 173% better (on average) than DCQCN and TIMELY. This improvement is not surprising as both DCQCN and TIMELY saturate at such heavy loads. Though the relative ordering with the light mix (Figure 13(b)) remains the same as that with the typical mix, the absolute throughputs are higher, as expected. We see the correspondence between Dart's median latency and throughput at high loads. Like the median latencies of DCQCN and TIMELY, their throughputs also differ slightly.

TABLE I
SHORT-FLOW PACKETS WITH ECN MARKS

Traffic mix / Load	Typical Mix			Light Mix				Heavy Mix	
	20	40	60	20	40	60	80	20	40
DCQCN	17	41	67	5	14	36	48	33	70
Dart	6	11	14	4	9	21	28	9	18

TABLE II
IOFD'S LOAD INCREASE AND PATH DILATION (%)

Traffic mix / Load	Typical			Light				Heavy	
	20	40	60	20	40	60	80	20	40
Load increase	3	6	9	0.4	0.8	1.4	3	11	15
Path dilation	8	15	23	4	8	14	26	15	21

Fall-Back to DCQCN: To evaluate DCQCN fall-backs in Dart, Table I shows the percent of short-flow packets with ECN marks under DCQCN and Dart. Because long flows are not latency-critical, we focus on short flows. As expected, both schemes incur more ECN marks as the load increases. However, Dart cuts the number of ECN marks by more than 4x at higher loads in typical and heavy mixes (i.e., significant fraction of short flows) where there is more congestion. These results (1) show that by filtering out receiver congestion and localized in-network congestion, Dart drastically reduces the number of DCQCN fall-backs and (2) reconfirm that these congestion components are significant.

Load Increase and Path Dilation Due to IOFD: Table II shows the percent increase in (a) network load and (b) short-flow path length under IOFD relative to DCQCN. Both the load and path dilation increase with more short flows ((i.e., light < typical < heavy) and at higher loads. For typical and heavy mixes, Dart increases the network load by 7% (geometric mean over the load settings) and dilates short-flow paths by 16% which is roughly one hop (our topology has 5.8 hops on average). Thus, Dart incurs a modest amount of network load to reduce congestion delays significantly.

C. Isolating Dart's Techniques

We quantify the relative contributions of Dart's two techniques: DASR and IOFD. Figure 14 plots Dart's 99^{th} percentile flow completion latency for the 8-KB short flows normalized to that of DCQCN (Y-axis) for the typical load mix (i.e., 40% short flows and 60% long flows) at various load levels (groups of bars along the X-axis). In addition to Dart, we quantify the benefits of DCQCN with priority queues using two priority levels to prioritize short flows over long flows (*Pri-Q*), IOFD without DASR (*IOFD-only*), DASR without IOFD or the look-ahead optimization in Section III-D (*DASR w/o LA*), and DASR with look-ahead but without IOFD (*DASR-only*).

As we see from Figure 14, Pri-Q does not improve latency at low loads where the long flows do not cause much congestion and hence provide limited opportunity. At higher loads, Pri-Q improves latency as expected. However, the improvement is limited because Pri-Q does not alleviate congestion *among* short flows, and, therefore, performs worse than IOFD-only and DASR-only, our key techniques. IOFD and DASR specifically address congestion among short flows — IOFD addresses localized congestion, whereas

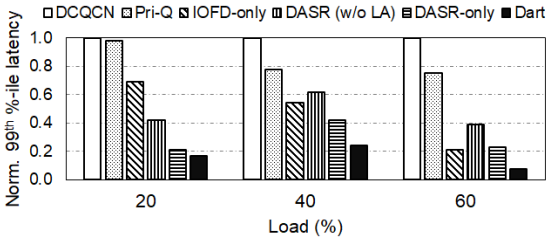

Fig. 14. Isolating Dart's techniques.

DASR addresses receiver congestion. Three key trends regarding the relative benefits of DASR and IOFD are apparent in Figure 14. First, at the intermediate load of 40% (middle bars), each of DASR (including the look-ahead) and IOFD contribute to Dart's improvements. DASR contributes more because receiver congestion is the common case (Section VI-B). Further, the difference between DCQCN and *IOFD-only* shows that IOFD can handle localized congestion without triggering DCQCN fall-back (Section IV-B). Second, at lower loads (left bars), most of the gains come from DASR which effectively protects the short flows from the long flows (79% of the 84% total latency reduction). This result is not surprising because in-network congestion is less likely at lower loads. The sizable difference between *DASR w/o LA* and *DASR-only* shows the look-ahead's impact. In the absence of localized congestion, the opportunity for IOFD is lower; as such *IOFD-only* contributes only 31% latency reduction in isolation, and approximately 21% incremental latency reduction over *DASR-only*. Finally, in contrast to the low-load results, IOFD contributes relatively more to the overall latency reduction at higher loads, where in-network congestion is more likely (79% of the 93% total latency reduction at 60% load). IOFD handles even this higher congestion without falling back to DCQCN. *DASR-only*'s relative contribution is smaller than IOFD's (77% latency reduction in isolation, and 35% incremental latency over *IOFD-only*). The median latencies follow the same trends. The effectiveness of *DASR-only* and *IOFD-only* illustrate the power of Dart's divide-and-specialize approach.

Sensitivity: We varied the deflection threshold (Section IV-B) as 5, 15 (default) and 20 KB. IOFD works well in the range of 5-15 KB, whereas the 20-KB threshold being close to the ECN threshold (22.5 KB) results in IOFD being disabled. We also varied the short-flow sizes as 2, 4 and 8 (default) KB. Dart's improvement across these flow sizes match those in Figure 11. Finally, we varied the incast degree as 6, 16 (default), and 26. At higher incast degree, Dart's latency improvement over DCQCN increases. However, at 60% load and incast degree of 26, the network saturates leaving no room for Dart. These results are not shown due to lack of space.

D. Comparison to RCP and ExpressPass

a) Comparison to RCP: We study RCP's convergence using an ns-3 implementation [49]. We simulate a simple topology with three servers that connect to a single switch. The topology uses 10 $Gbps$ links with a delay of 5 μs, matching today's datacenters. We set RCP's main parameters, $\alpha = 0.4$ and $\beta = 0.4$, as recommended [29], [50].

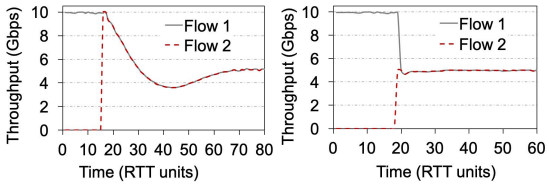

Fig. 15. Convergence time for RCP and Dart.

We compare how fast RCP and Dart converge to the fair-share rate of 5 $Gbps$ for the simple case of receiver congestion with two flows. In our experiment, the second flow starts after the first flow. Figure 15 shows time, in RTT time units, along the X axis and the throughputs achieved by the two flows for RCP (in Figure 15(a)) and Dart (in Figure 15(b)), along the Y axis. Because RCP does not *explicitly* count flows, it requires about 60 $RTTs$ to converge (i.e., the second flow joins at about 15 RTTs but the rate does not converge to 5 $Gbps$ until 75 RTTs). In contrast, Dart converges in 1 RTT by *explicitly* tracking the number of senders at the receiver.

b) Comparison to ExpressPass: We obtained the ExpressPass [31] simulator from the authors. We found that while ExpressPass converges in 4 RTTs for two senders and two receivers in a simple dumbbell topology (matches ExpressPass paper's results), it takes 20 RTTs for 10 senders and one receiver in the fat tree topology used in the paper (the paper does not show this case). In the former case, convergence is effected by fair queuing of credit packets at the switch where the two flows collide, whereas the latter case converges using ExpressPass's BIC-TCP-like algorithm, which is iterative and slow.

VII. Related Work

Because we have discussed DCQCN, TIMELY, NUM-Fabric, ExpressPass, and NDP at length in earlier sections, we focus on other work related to our key techniques – DASR (congestion control) and IOFD (load balancing).

DCTCP [9], a pioneering work in datacenter transport protocols, finely modulates the sending rate by observing ECN marks in each RTT and nearly eliminates incast-induced timeouts. D^2TCP [51] builds upon DCTCP to prioritize flows based on deadlines. TCP Bolt [48] uses flow-level congestion control via ECN to address PFC's limitations. ICTCP [52] iteratively adjusts the TCP receive window before incast-induced packet drops. Like DCQCN and TIMELY, all these TCP variants incur several RTTs (i.e., tens or hundreds) to converge to the appropriate sending rate. In RCP [29], another pioneering work, routers explicitly convey the fair share rate to the senders that share a link. However, because RCP routers don't have per-flow state and many short flows begin and end in each RTT, RCP's convergence is iterative and takes many RTTs. D3 [53] and PDQ [54] employs explicit rate control to prioritize critical flows. For the same reasons as RCP, D3 and PDQ's convergence requires several RTTs. pFabric [47], Karuna [55], UPS [56] and pHost [57] address flow scheduling but their rate control is still iterative. EyeQ [15] leverages RCP to provide weighted fair share in multi-tenant datacenters but inherits RCP's iterative convergence. QCN [58] provides end-to-end congestion control for RDMA in Layer2 but not in IP-switched datacenter-scale networks. IRN [40] adapts ideas from I-WARP to avoid the problems associated with PFC. Unlike DASR, none of the above work isolates receiver congestion to achieve fast, *one-RTT* convergence. DIATCP [59] is a receiver-based rate control protocol where each sender informs the receiver of the message size and deadline so that the receiver calculates the per-sender sending rate accordingly. Though DIATCP can achieve one-RTT convergence, its calculation requires an a priori globally-common RTT which is achieved by artificially delaying *acks*. In a DC setting, such delay would force *all* flows to have the same RTT as the tail, which is often 2-5x longer than the median [9]. In contrast, Dart sends only the sender count to the senders each of which then flexibly and locally arrives at a sending rate based on the currently-observed, individual RTT. Apart from this fundamental difference, while both Dart and DIATCP use timeouts (Section III-B) they do so for different purposes. Because RDMA has explicit message start/end markers, Dart does not need to use timeouts for message ends. Instead, Dart uses timeouts for infrequent, catastrophic events like crashes. In contrast, DIATCP uses timeouts to detect idling because TCP does not have message start/end markers. Thus, Dart's mechanisms are optimized for RDMA unlike DIATCP. Because flow idling is more frequent than crashes, DIATCP's timeouts can affect performance – a timeout being too short means delayed flow completions and being too long means wasted throughput. Further, each sender in DIATCP provides its deadline and message size to the receiver at the time of establishing the connection. This exchange occurs before each flow starts, but not necessarily before the start of all the other flows in an incast group. As such, DIATCP cannot know the correct rate before *all* the flows start. In contrast, DASR's lookahead (Section III-D) is based on the application where all the flows in an incast group know before *any* of them start that they are a part of an incast group. Thus, DIATCP does not achieve DASR's lookahead.

Among load balancing schemes, MPTCP [21], [60] splits a TCP flow into many sub-flows that may be routed independently along different paths. FlowBender [25] proposes re-hashing at end-hosts to change flow paths. Presto [26] splits large flows into equal-sized flowcells and uses a central scheduler to balance the load. DeTail [24], Random Packet Spraying [23], and DIBS [28] balance load at the finer granularity of packets. Recent schemes [61], [62] improve load balancing but they also require reordering at the receiver. In contrast to these above schemes all of which reorder packets which is not supported by RDMA, IOFD is designed to deflect without packet reordering. SPAIN [22] and CONGA [27] also avoid packet reordering. However, SPAIN pre-computes multiple paths which are mapped to different VLANs but such precomputation may be slow in reaction to short flows in a datacenter. CONGA uses global congestion information to load balance at the granularity of flowlets. However, CONGA works only with two-tier leaf-spine topology and does not scale to large datacenters.

VIII. Conclusion

RDMA can significantly reduce datacenter network latencies compared to TCP but provides suboptimal end-to-end congestion control for the well-known problem of incasts.

Previous schemes target the full generality of the congestion problem and rely on slow, iterative convergence to the appropriate sending rates. Several papers have shown that even in oversubscribed datacenter networks most congestion occurs at the receiver. Accordingly, we proposed a divide-and-specialize approach, called *Dart*, which isolates the common case of receiver congestion and further sub-divides the remaining in-network congestion into the simpler spatially-localized and the harder spatially-dispersed cases. To address receiver congestion, we proposed *direct apportioning of sending rates (DASR)* in which a receiver for n senders directs each sender to cut its rate by a factor of n. DASR converges in only one RTT. For the spatially-localized case, Dart adds novel switch hardware for *in-order flow deflection (IOFD)* because RDMA disallows packet reordering on which previous load balancing schemes rely. IOFD provides fast (under one RTT), light-weight response. For the uncommon spatially-dispersed case, Dart falls back to DCQCN. Our small-scale testbed measurements showed that Dart converges in one RTT and achieves 60% (2.5x) lower tail (99^{th}-percentile) latency than and similar throughput as InfiniBand. Our at-scale simulations showed that Dart achieves 79% (4.8x) lower tail latency, and 58% higher throughput than TIMELY and DCQCN. As datacenter networks evolve towards adopting RDMA to avoid TCP's overhead, Dart's superior latency and throughput characteristics are likely to be attractive.

REFERENCES

[1] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over infiniband," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, Jun. 2004, doi: 10.1023/B:IJPP.0000029272.69895.c1.

[2] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proc. 11th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 429–444. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616488

[3] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*. Berkeley, CA, USA: USENIX Association, 2013, pp. 103–114. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535475

[4] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Berkeley, CA, USA: USENIX Association, Apr. 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=208395

[5] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proc. ACM Conf. (SIGCOMM)*. New York, NY, USA: ACM, 2014, pp. 295–306, doi: 10.1145/2619239.2626299.

[6] J. Jose *et al.*, "Scalable memcached design for infiniband clusters using hybrid transports," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (Ccgrid)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 236–243, doi: 10.1109/CCGrid.2012.141.

[7] E. G. Gran *et al.*, "First experiences with congestion control in infiniband hardware," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, pp. 1–12.

[8] D. Crupnicoff, S. Das, and E. Zahavi, "White paper: Deploying quality of service and congestion control in infiniband-based data center networks," Mellanox Technol., Sunnyvale, CA, USA, Tech. Rep. 2379, Nov. 2005.

[9] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.* New York, NY, USA: ACM, 2010, pp. 63–74, doi: 10.1145/1851182.1851192.

[10] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.* New York, NY, USA: ACM, 2008, pp. 63–74, doi: 10.1145/1402958.1402967.

[11] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 892–901, Oct. 1985. [Online]. Available: http://dl.acm.org/citation.cfm?id=4492.4495

[12] Y. Zhu *et al.*, "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2015, pp. 523–536, doi: 10.1145/2785956.2787484.

[13] R. Mittal *et al.*, "Timely: RTT-based congestion control for the datacenter," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2015, pp. 537–550, doi: 10.1145/2785956.2787510.

[14] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas. (IMC)*. New York, NY, USA: ACM, 2010, pp. 267–280, doi: 10.1145/1879141.1879175.

[15] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," in *Proc. 10th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 297–312.

[16] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. Internet Meas. Conf. (IMC)*. New York, NY, USA: ACM, 2017, pp. 78–85.

[17] M. Handley *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2017, pp. 29–42.

[18] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2015, pp. 123–137.

[19] A. Singh *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2015, pp. 183–197.

[20] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. (IMC)*. New York, NY, USA: ACM, 2009, pp. 202–208.

[21] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*. New York, NY, USA: ACM, 2011, pp. 266–277, doi: 10.1145/2018436.2018467.

[22] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "Spain: Cots data-center Ethernet for multipathing over arbitrary topologies," in *Proc. 7th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2010.

[23] A. Dixit, P. Prakash, Y. Hu, and R. Kompella, "On the impact of packet spraying in data center networks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 2130–2138.

[24] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun.* New York, NY, USA: ACM, 2012, pp. 139–150, doi: 10.1145/2342356.2342390.

[25] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, 2014, pp. 149–160.

[26] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 465–478.

[27] M. Alizadeh *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM Conf. (SIGCOMM)*, 2014, pp. 503–514.

[28] K. Zarifis, R. Miao, M. Calder, E. Katz-Bassett, M. Yu, and J. Padhye, "Dibs: Just-in-time congestion mitigation for data centers," in *Proc. 9th Eur. Conf. Comput. Syst. (EuroSys)*. New York, NY, USA: ACM, 2014, pp. 6:1–6:14, doi: 10.1145/2592798.2592806.

[29] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown, "Processor sharing flows in the Internet," in *Proc. 13th Int. Conf. Qual. Service (IWQoS)*. New York, NY, USA: Springer-Verlag, 2005, pp. 271–285.

[30] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "Numfabric: Fast and flexible bandwidth allocation in datacenters," in *Proc. Conf. ACM SIGCOMM Conf. (SIGCOMM)*, 2016, pp. 188–201.

[31] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2017, pp. 239–252.

[32] J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1320–1331, Dec. 1993, doi: 10.1109/71.250114.

[33] P. Baran, "On distributed communications networks," *IEEE Trans. Commun. Syst.*, vol. 12, no. 1, pp. 1–9, Mar. 1964.

[34] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013, doi: 10.1145/2408776.2408794.

[35] D. Dunning *et al.*, "The virtual interface architecture," *IEEE Micro*, vol. 18, no. 2, pp. 66–76, Mar./Apr. 1998, doi: 10.1109/40.671404.

[36] (2017). *Infiniband Trade Association*. [Online]. Available: http://www.infinibandta.org

[37] (2017). *RDMA Over Converged Ethernet*. [Online]. Available: http://www.mellanox.com/page/products_dyn?product_family=79

[38] *Agilio CX SmartNICS*. Accessed: Dec. 28, 2019. [Online]. Available: https://www.netronome.com/products/agilio-cx

[39] *Mellanox Connectx-5 Product Brief*. Accessed: Dec. 28, 2019. [Online]. Available: http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_VPI_Card.pdf

[40] R. Mittal *et al.*, "Revisiting network support for RDMA," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2018, pp. 313–326, doi: 10.1145/3230543.3230557.

[41] C. Clos, "A study of non-blocking switching networks," *Bell Labs Tech. J.*, vol. 32, no. 2, pp. 406–424, 1953.

[42] L. Gao, "On inferring autonomous system relationships in the Internet," *IEEE/ACM Trans. Netw.*, vol. 9, no. 6, pp. 733–745, Dec. 2001.

[43] F. Silla and J. Duato, "Improving the efficiency of adaptive routing in networks with irregular topology," in *Proc. 4th Int. Conf. High-Perform. Comput.*, Dec. 1997, pp. 330–335.

[44] S. Hu *et al.*, "Tagger: Practical PFC deadlock prevention in data center networks," *IEEE/ACM Trans. Netw.*, vol. 27, no. 2, pp. 889–902, Apr. 2019.

[45] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, vol. C-36, no. 5, pp. 547–553, May 1987.

[46] S. Hu *et al.*, "Deadlocks in datacenter networks: Why do they form, and how to avoid them," in *Proc. 15th ACM Workshop Hot Topics Netw. (HotNets)*. New York, NY, USA: ACM, 2016, pp. 92–98, doi: 10.1145/3005745.3005760.

[47] M. Alizadeh *et al.*, "pFabric: Minimal near-optimal datacenter transport," in *Proc. ACM SIGCOMM Conf. SIGCOMM*. New York, NY, USA: ACM, 2013, pp. 435–446, doi: 10.1145/2486001.2486031.

[48] B. Stephens, A. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter, "Practical DCB for improved data center networks," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 1824–1832.

[49] M. Flores, A. Wenzel, and A. Kuzmanovic, "Enabling router-assisted congestion control on the Internet," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[50] N. Dukkipati, "Rate control protocol (RCP): Congestion control to make flows complete quickly," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, USA, 2008.

[51] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2012.

[52] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data center networks," in *Proc. 6th Int. Conf. (Co-NEXT)*. New York, NY, USA: ACM, 2010, pp. 13:1–13:12, doi: 10.1145/1921168.1921186.

[53] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*. New York, NY, USA: ACM, 2011, pp. 50–61, doi: 10.1145/2018436.2018443.

[54] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*. New York, NY, USA: ACM, 2012, pp. 127–138, doi: 10.1145/2342356.2342389.

[55] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with karuna," in *Proc. Conf. ACM SIGCOMM Conf. (SIGCOMM)*, 2016, pp. 174–187.

[56] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proc. 13th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 501–521.

[57] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "pHost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2015, pp. 1:1–1:12.

[58] (2007). *QCN: Quantized Congestion Notification an Overview*. [Online]. Available: http://www.ieee802.org/1/files/public/docs2007/au_prabhakar_qcn_overview_geneva.pdf

[59] J. Hwang, J. Yoo, and N. Choi, "Deadline and incast aware TCP for cloud data center networks," *Comput. Netw.*, vol. 68, pp. 20–34, Aug. 2014.

[60] C. Raiciu *et al.*, "How hard can it be? Designing and implementing a deployable multipath TCP," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, San Jose, CA, USA, 2012.

[61] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 253–266.

[62] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 225–238.

Jiachen Xue received the bachelor's degree in computer engineering from Beihang University in 2006, the M.S. degree in electrical engineering from Arizona State University in 2008, and the Ph.D. degree in computer engineering from Purdue University in 2017. He currently works as a Senior Distributed System Engineer at NVIDIA.

Muhammad Usama Chaudhry received the bachelor's degree from the National University of Computer Science, Islamabad, Pakistan, in 2014, and the M.S. degree from the Computer Science Department, University of Illinois at Chicago, in 2019. He is currently a Software Engineer with VMware Inc.

Balajee Vamanan received the Ph.D. degree from Purdue University in 2015. Prior to his Ph.D., he worked at NVIDIA as a Design Engineer. He is currently an Assistant Professor with the Department of Computer Science, University of Illinois at Chicago (UIC). His research interests span various aspects of computer networks and computer systems.

T. N. Vijaykumar is currently a Professor with the School of Electrical and Computer Engineering, Purdue University. His research interests include computer architecture targeting on various aspects of performance, power, programmability, and reliability of computer hardware and systems. Recognition of his work has received awards including the 1999 NSF CAREER Award, the IEEE Micro's Top Picks on computer architecture papers in 2003 and 2005, being listed in the International Symposium on Computer Architecture (ISCA) Hall of Fame, and the First Prize in 2009 Burton D. Morgan Business Plan Competition.

Mithuna Thottethodi received the B.Tech. degree (Hons.) in computer science and engineering from the Indian Institute of Technology Kharagpur and the Ph.D. degree in computer science from Duke University. He is currently an Associate Professor of electrical and computer engineering with Purdue University. His research interests include computer architecture, distributed systems, and networks. He received the NSF CAREER Award in 2007.