Burst-tolerant Datacenter Networks with Vertigo

Sepehr Abdous*, Erfan Sharafzadeh*, Soudeh Ghorbani
Johns Hopkins University

ABSTRACT

Microsecond-scale congestion events, known as *microbursts*, are a main cause of packet loss and poor application performance in today's datacenters. Given the low network utilization in datacenters, one would expect packet *deflection*, in-situ re-routing of packets that arrive at a full buffer to a different port, to effectively prevent packet loss. However, if deployed naively, deflection leads to excessive packet re-ordering, exacerbated congestion, and head-of-the-line blocking in switch buffers. In this study, we resolve the above challenges by selectively deflecting the packets that cause persistent congestion in the network. To enable this, we augment the end-host network stacks with a transport-independent extension that tracks and marks flows with their remaining bytes. Our in-network deflection component uses the flow size information to re-route packets from flows with more data to send. Finally, an extension to the receive-side of end-host stacks retrieves the correct ordering of packets before passing them to transport and higher-level protocols. We evaluate our design, Vertigo, under diverse datacenter workloads and show that it is effective in managing microbursts under light and heavy loads and when combined with various congestion control algorithms. For example, in a leaf-spine network under 85% load, Vertigo reduces the mean incast query completion times by 3.5×, 3.3×, 5× compared to ECMP, DRILL, and DIBS when using TCP, 3×, 3.5×, 4.5× alongside DCTCP, and 43×, 33×, 16× when using Swift, respectively.

CCS CONCEPTS

• **Networks → Data center networks**.

KEYWORDS

Datacenter networks, Microbursts, Packet deflection

ACM Reference Format:
Sepehr Abdous*, Erfan Sharafzadeh*, Soudeh Ghorbani. 2021. Burst-tolerant Datacenter Networks with Vertigo. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3485983.3494873

*Equal contribution. A coin toss decided the order of the first two authors.

1 INTRODUCTION

Driven primarily by two trends—the disaggregation of storage, compute, and memory across the network for cost savings and the rising demand for high-speed transmission in new technologies and applications [47, 48]—datacenters today have exceedingly stringent low-latency requirements. To enable resource disaggregation, remote resources (GPU, memory, disk, etc.) should be accessible over the network within 3-5μs [48]. In emerging applications and technologies such as NVMe (non-volatile memory express) and large-scale machine learning workloads, the network is frequently the performance bottleneck because their storage and computation resources are extremely fast [48]. Despite significant progress towards building ultra-low-latency datacenter networks in recent years, why does the network continue to remain the performance bottleneck? A key challenge is reportedly *microbursts*, short-lived periods of congestion that last for less than a millisecond and cause the majority of packet loss in datacenters [15, 48, 76]. The extreme packet drops caused by microbursts lead to re-transmissions that impose significant latency and degrade application performance. Managing microbursts is challenging because of their short lifespans and their diverse and ever-changing root causes (applications, TCP artifacts such as ACK compression, offloading features in NICs, *etc.* [6, 8, 40, 41, 44, 51, 53, 73, 76]). Rack-level traffic measurements at Facebook, for instance, show that more than 70% of microbursts last for less than a few tens of microseconds, significantly shorter than the frequency of most deployed measurement frameworks [76].

Given by the necessity of reacting to microbursts in situ and in real-time, a group of proposals attempts to manage microbursts in the network core, *e.g.,* via balancing the load among multiple shortest paths [33], deflecting the excess load across neighboring switches with spare capacity [65, 75], and provisioning enough buffer space in switches to absorb bursts [13]. Although effective for small scales and lightly loaded networks, **network-centric techniques fail under load and at scale.** Load balancing and buffer sizing techniques, for example, cannot manage large-scale incasts where a large number of hosts simultaneously transmit data to a single receiver [47], because the intensity of the burst in such cases exceeds the buffer capacity of any single datacenter switch (commonly shallow buffered) and the number of paths that the burst can be distributed among. Similarly, deflecting packets extends their path lengths, *e.g.,* by 20% under 50% offered load (§2), and increases utilization which exacerbates the congestion if the network is already overloaded. By preventing packet drops, deflection further delays informing the hosts that they need to throttle their send rate. Our experiments show that under 80% load, random deflection completes 10× fewer incast queries and the average flow completion times is 45% higher than a simple baseline. Datacenters today experience a wide range of workloads, scale, and utilization, including frequent epochs of high utilization and extreme-scale incast events (thousands of flows arriving simultaneously at the same destination) [43, 47, 48]. For a burst management technique to be viable, it is an essential necessity that it handles extreme load gracefully.

On the other hand, given the greater visibility and control over the sources of traffic at the edge, another group of host-centric designs strives to proactively identify and prevent the formation of microbursts at the edge, sometimes with some feedback from the network (*e.g.,* regarding the queueing delay [48] and congestion [6]). Adding jitter in the application layer to prevent synchrony [29], credit-based transport protocols that coordinate and schedule the flows sent to a receiver [30, 35, 56], and feedback-based congestion control protocols that strive to impede bursts and their subsequent packet loss by reacting to increasing RTTs and network congestion faster [6, 47, 48, 63] are all examples of designs in which the hosts play a central role in preventing the formation of microbursts. Compared to core-centric designs, **host-based techniques are fundamentally limited by their slower reaction to microbursts, typically an RTT or slower.**

We advocate co-designing the networking layer in the core and the edge to make it burst-tolerant. Given that microbursts are extremely short-lived, the network core should be capable of handling them in real-time and in place, *e.g.,* by distributing microburst packets across the network. However, to remain efficient under various degrees of load, the network should distinguish between microbursts and an overall high degree of load and treat each differently. For example, while *deflecting* the packets of a local, transient burst to other switches when the overall utilization is low improves flow completion times, *dropping* them, when the overall load is high, helps reduce the congestion and improves performance. Making such distinctions can be greatly facilitated with some assistance from the edge. We show that a simple and efficient extension to senders' networking stack enables such discretion in the network core by tagging packets with flow size information. Similarly, on the receiver-side, a re-sequencing shim layer can retrieve the correct ordering of packets and thus shield the transport and application protocols from the excessive reordering caused by deflection.

To realize this vision, we design Vertigo, an edge-core co-design of the networking layer that leverages end-host knowledge of the flow size information to selectively deflect and drop packets based on their likely contribution to persistent congestion (as opposed to microbursts). Vertigo consists of three components deployed on the path of a datacenter packet. (1) We design an extension to the end-hosts network stack that tracks and tags every packet inside a flow with the remaining bytes of its flow, referred to as RFS (Remaining Flow Size)[1]. A boosting module decreases the RFS fields of re-transmitted packets to ensure they do not starve. (2) In the network core, when a switch receives a packet and the output queue of the packet is full, the switch selects the packet with the largest RFS among the newly arrived packet and the packets in its output queue to deflect: the switch then randomly selects two queues and inserts the packet into the least loaded one. If both queues are full, the switch randomly selects one of them and drops the packet with the largest RFS among the enqueued packets and the deflected packet in order to keep the packets with the lowest RFS in the queue. (3) Finally, we design a transport-independent packet ordering framework on the receiver hosts' RX path that detects and buffers out-of-order packets to wait for packets experiencing prolonged RTTs due to deflection. Packets of the flows with more bytes to send (*i.e.,* those with large RFS) are more likely to contribute to persistent congestion. When facing microbursts, Vertigo prioritizes such packets for deflecting (when a randomly selected queue in the switch has spare capacity) and for dropping (when the network is congested, indicated by two randomly selected queues being *both* full at the same time). This enables Vertigo to gracefully handle microbursts even under extreme loads.

Vertigo is a burst-tolerant, fast L2/L3 routing technique for datacenters. Albeit more efficient than its L2/L3 counterparts, it still only provides a best-effort reachability service that may drop and reorder packets. Thus, it should be deployed below the appropriate transport protocols that provide higher-level services such as congestion control (*e.g.,* to throttle the send rate when the overall send rate is higher than the capacity of the network), loss recovery (due to congestion and failures), and fairness. In our evaluations, we test the performance of Vertigo as the substrate running below TCP, DCTCP, and a state-of-the-art datacenter congestion control algorithm, Swift [47]. Vertigo consistently outperforms the other L2/L3 baselines such as DIBS [75] (a representative of deflection routing in datacenters), DRILL [33] (a microburst-tolerant load balancer), and ECMP (the most widely deployed forwarding protocol in datacenters) for all these transport protocols, especially under load and extreme incast scales. For example, in a network with 55% overall link utilization and a bursty workload, Vertigo+DCTCP reduces mean incast query completion times (QCT) by 48%, and 58% over DRILL+DCTCP and DIBS+DCTCP, respectively. Under 85% load, incast queries finish 47% and 68% faster under Vertigo+DCTCP compared to DRILL+DCTCP and DIBS+DCTCP. Under the same load (85%), Vertigo+Swift improves the QCT of DRILL+Swift and DIBS+Swift by a factor of 32 and 15, respectively.

At its core, Vertigo is a deflection routing technique that harnesses hosts' visibility into the workload to remain efficient under extreme scales and loads. We demonstrate why a naive implementation of deflection quickly fails in typical datacenter scenarios (§2), outline our design for making deflection practical in datacenters (§3), and evaluate the effectiveness of Vertigo in typical datacenter scenarios, including under various degrees of load, traffic burstiness, and two datacenter topologies, via extensive simulations and microbenchmarking (§4).

2 DEFLECTION ROUTING IN DATACENTERS: CHALLENGES AND OPPORTUNITIES

Three factors—(a) the prevalence of microsecond-scale congestion periods, (b) low overall utilization, and (c) shallow buffer switches—make datacenters amenable to deflection routing. In deflection routing, when the output buffer of the preferred path (typically the shortest path) of a packet is congested, instead of dropping the packet, the router detours it to a neighboring router. This distributes the load of the hotspots across the network and reduces the need for having deep buffers in individual routers.

[1] In §4, we show that Vertigo continues to be more effective than other baselines such as ECMP and DIBS [75] even when flow size information is not available in advance. However, having this information improves its average incast query completion time by 13%.

Deflection routing and *hot-potato* routing, a variant of it in which the router is assumed to be buffer-less,[2] are commonly deployed in networks where packet buffers are scarce and expensive such as optical networks [17–19, 21, 34, 38, 45] and networks-on-chip [26, 50].

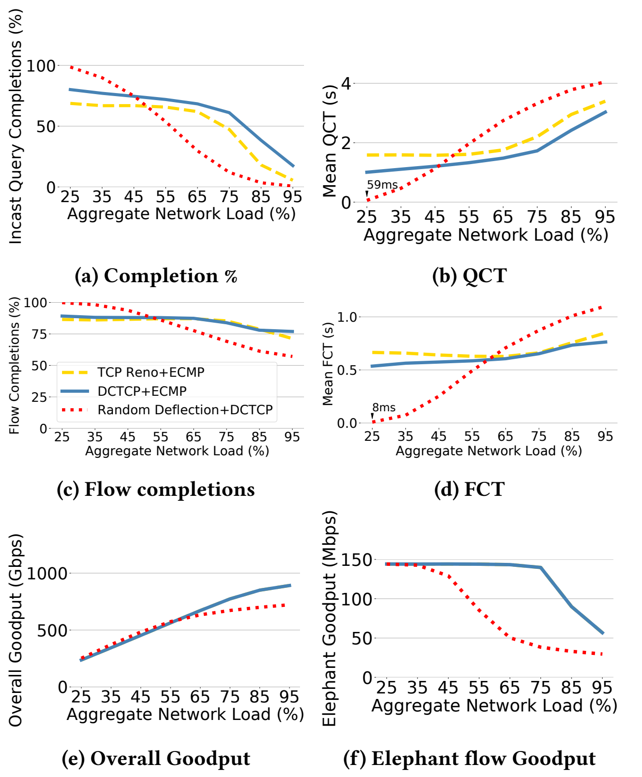
Due to the prohibitive cost and adverse impact of large-buffer switches on latency, akin to optical networks, datacenter switches commonly have shallow buffers. Plus, congestion and packet loss usually happen when there is spare capacity elsewhere in the network. Combined, these characteristics warrant an investigation of packet deflection in datacenters. With extensive simulations, we demonstrate that, in datacenters, albeit promising in lightly loaded networks, deflection routing quickly fails under load, *e.g.,* under 75% load, deflection routing completes 3× *fewer* application queries and results in 1.87× *higher* query completion times compared to a naive baseline (ECMP+vanilla TCP). Plus, even under low load, it can cause excessive packet drops and reordering that is problematic for pervasive transport protocols, applications, and network monitoring and diagnosis systems that interpret such events as symptoms of network failure [9, 10, 77], *e.g.,* under 35% offered load, random deflection leads to a 10-fold increase in packet re-ordering at the receiving transport and 57% raise in packet loss compared to ECMP.

In our experiments, unless otherwise specified, we deploy DIBS [75], a recent deflection routing technique for datacenters, as a representative of deflection routing. Concretely, with DIBS, when a switch receives more packets than it can enqueue in the output queue or forward, it detours the excess packets to a randomly selected port with enough buffer space. DIBS relies on DCTCP for congestion control and it disables DCTCP's fast retransmit mechanism to prevent the side-effects of packet re-ordering. We next discuss the problems that random deflection creates and outline an approach to make it practical.

Deflection quickly fails under load. Packet loss is one of the main signals of congestion that congestion control algorithms rely on. Preventing drops in the network core by deflecting the packets delays sending this signal to the sources of traffic. Plus, the deflection itself increases the overall utilization—further exacerbating the congestion. The traffic, therefore, continues to flow to the point where many buffers are saturated, and many packets are inevitably lost, imposing a significant spike in latency.

To quantify the problem, we simulate a leaf-spine network with 4 cores, 8 aggregates, and a total of 320 servers connected via 10Gbps links to ToR switches with 300KB per-port buffer capacity. We deploy TCP Reno, DCTCP, and DIBS, and write an incast application in which a set of randomly selected clients periodically send queries to 100 randomly selected servers at predefined intervals. Upon receiving a query, each server responds with 40KB of data, and the initiator marks the query as completed after all 100 replies are received. We gradually increase the overall offered load by lowering incast event inter-arrivals in a network filled with 15% background load (the flow sizes and interarrival times for the background load are from [62]). Applying random packet deflection as proposed in DIBS increases the average number of hops that packets traverse by 20%. As Figure 1 shows, under 80% load, this results in 54% *higher* mean query completion times and 30% *higher* flow completion time tail at 99th percentile compared to ECMP+DCTCP. The results also show that the back-off behavior by random deflection starts to manifest as the aggregate load passes 65%. After this point, increasing the load results in only a negligible improvement in the application goodput with random deflection (Figure 1e), and a substantial drop in the goodput for elephant flows (flows larger than 10MB) as depictd in Figure 1f. *These results highlight the need for a smart, load-adaptive deflection technique.*

[2]Hot-potato routing in this context should not be confused with hot-potato routing in BGP where, between multiple equally good BGP routes, a router selects the one with the closest egress point [70].



(a) Completion %

(b) QCT

(c) Flow completions

(d) FCT

(e) Overall Goodput

(f) Elephant flow Goodput

Figure 1: Random packet deflection starts to break as the aggregate network load passes 65% due to excessive RTOs caused by packet drops. The workload consists of 15% background traffic and varying incast query arrivals.

Deflection leads to large delays in completing mice flows. Preventing packet drops allows the transmission windows of large flows to grow. Large flows contribute to long-lasting congestion in switch buffers, and their bursty behavior may result in head-of-the-line blocking for short flows. Plus, the amortized added delay of taking longer routes is higher for mice flows. Our simulations show that random deflection increases the average queueing time of mice flows (< 100KB) by 111% and increases their average flow completion time (FCT) by 40%. *These results suggest that longer flows should be prioritized for deflection.*

Deflection results in excessive packet reordering. Reordering can reduce throughput and increase flow completion times. In most TCP variants (including DCTCP), for example, three consecutive duplicate ACKs (resulted from out-of-order delivery) triggers the fast retransmit mechanism where the transmission window is divided in half. To prevent this, some prior techniques, including

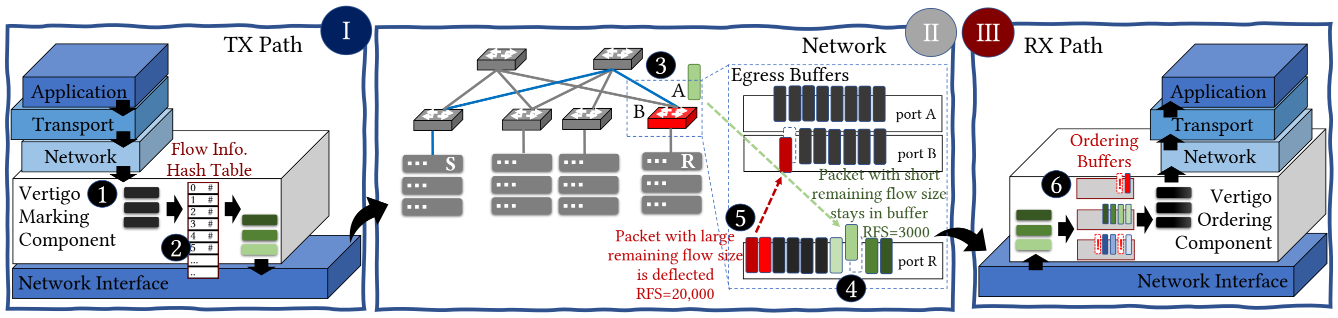


Figure 2: An illustration of a scenario where a microburst causes the last-hop buffer to overflow. Vertigo absorbs the burst by selectively deflecting packets belonging to flows that contribute to long-lasting congestion.

DIBS, disable the fast retransmit phase [65, 75]. However, when the switch has to drop a packet, perforce (*e.g.,* in extreme-scale incasts or in globally congested networks), disabling fast retransmit increases the delay as every dropped packet will require a retransmission timeout (RTO) to be retransmitted. This is generally much slower than the fast retransmit. Our simulation results demonstrate that random deflection increases the packet re-ordering by a factor of 3, leading to 18% reduction in the overall throughput of large flows under 95% load. Plus, network monitoring and diagnosis systems interpret retransmissions and reorderings as signals of network failure [9, 10, 77]. *These results underscore the necessity of re-sequencing the packets immediately after they arrive at the destination.*

Deflection causes numerous packet drops even in lightly loaded networks. Deflecting packets to a randomly selected neighboring switch may create congestion in that switch. We verify this by comparing random deflection vs. a load balancing technique inspired by the "power of two choices" paradigm [55] where for deflecting each packet, we randomly sample two queues and send the packet to the one with lower queue occupancy. In a lightly loaded network with 35% overall link utilization, random deflection results in 54.5% higher packet loss compared to the power of two choices technique. *These results point to an opportunity for balancing the deflected packets evenly in the network.*

We next discuss how we overcame the hurdles of random deflection to build an efficient deflection technique.

3 VERTIGO: TIMELY REACTION TO MICROBURSTS

We present the design of Vertigo, a fast and simple solution to microbursts based on packet deflection. Vertigo is composed of extensions to the host network stack and an in-network deflection and scheduling component. In Vertigo, the sender host marks the outgoing packets with the remaining bytes of the flow. In the network core, Vertigo switches harness this information to selectively deflect packets more likely to contribute to long-lasting congestion when microbursts occur (or to prioritize dropping these packets when the overall load is high). Finally, Vertigo's ordering component, deployed on the receive path at end-hosts, ensures that packets arrive at the transport and higher layers in the correct order that they have been sent.

An Illustrative example. Figure 2 presents an operational overview of Vertigo when the Top-of-Rack (ToR) switch buffer facing the destination host has no room to accommodate the incoming packets.[3] The marking component on the TX path, deployed as an independent extension to the existing network stack, receives the packets from the upper layers ❶ and marks them with the Remaining Size of the Flow (*RFS*) in bytes (*e.g.,* for the last packet of a flow, RFS is equal to packet's payload length) ❷. This information will enable Vertigo's in-network component to make accurate deflection, dropping, and scheduling decisions based on the Shortest Remaining Processing Time (SRPT) paradigm [7] by implementing priority-based buffering primitives in the network core (§3.1). Even though scheduling flows based on their remaining processing times does not guarantee optimal performance, it is shown to produce near-optimal results when deployed in conjunction with commonly-used congestion control techniques such as TCP or DCTCP [7, 58]. Our results in §4.3 illustrate that Vertigo's use of SRPT paradigm improves the average query completion time by 75% under 95% load. In the network, each switch places the packet in a priority output queue, sorted in ascending order of RFS. That is, packets of the flows with lower remaining sizes will be transmitted first. The packet is forwarded through the network until it reaches a full buffer (the destination ToR in this example) ❸. Inserting the packet in a full priority queue results in buffer overflow and the packet with the largest RFS being dequeued[4]. In this example, this results in the green packet (RFS=3000) being enqueued ❹ and the red packet (RFS=20000) being popped (§3.2). The dequeued packet is a candidate for deflection: the switch randomly selects two output ports and enqueues the packet to the least loaded one ❺.[5] Finally, at the destination host, the same RFS field helps Vertigo's ordering component to detect and re-shuffle the out-of-order packets before passing them to upper layers ❻ (§3.3).

The visibility provided by the sender's network stack allows for a simple and efficient way to distinguish between persistent congestion and microbursts without imposing the computational and storage overhead of doing so to the network core, all while

[3]This is the most common case of packet drops in datacenters; most of the packet loss (*e.g.,* 90% in a Facebook datacenter [76]) occur in the ToR-server direction [68, 76].
[4]In practice, more than one packet may be removed from the queue because of different packet sizes. For simplicity, in this illustrative example, we assume that all packets are identically sized.
[5]If both queues are full, we randomly select one and insert the deflected packet. This will result in a packet loss but full queues in both the forward and deflection paths signal severe congestion in the network. Therefore, Vertigo does not prevent packet drops in this case.

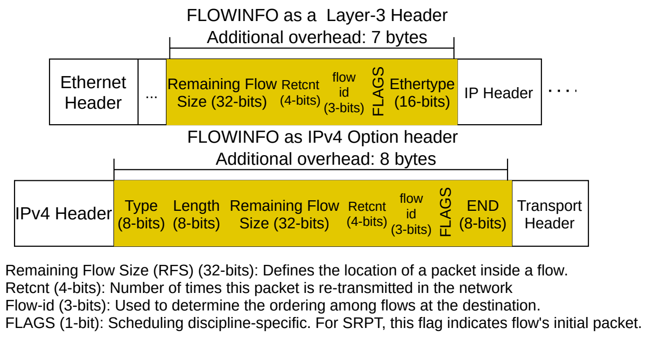

FLOWINFO as a Layer-3 Header
Additional overhead: 7 bytes
Ethernet Header
...
Remaining Flow Size (32-bits)
Retcnt (4-bits)
flow id (3-bits)
FLAGS
Ethertype (16-bits)
IP Header
....
FLOWINFO as IPv4 Option header
Additional overhead: 8 bytes
IPv4 Header
Type (8-bits)
Length (8-bits)
Remaining Flow Size (32-bits)
Retcnt (4-bits)
flow id (3-bits)
FLAGS
END (8-bits)
Transport Header
Remaining Flow Size (RFS) (32-bits): Defines the location of a packet inside a flow.
Retcnt (4-bits): Number of times this packet is re-transmitted in the network
Flow-id (3-bits): Used to determine the ordering among flows at the destination.
FLAGS (1-bit): Scheduling discipline-specific. For SRPT, this flag indicates flow's initial packet.

Figure 3: Two implementations of the flowinfo header. On top, the flowinfo header is implemented as a layer-3 header that encapsulates the IP header. On the bottom, the flowinfo is implemented inside IPv4 options header.

enabling transport-independent packet ordering at the destination. Following the path of a fresh packet, we describe each component in more detail.

3.1 TX Path: Marking Component

3.1.1 Priority Framework. Advance knowledge of flow sizes allows the datacenter network designers to implement scheduling policies that resemble the shortest remaining processing time (SRPT) discipline and therefore achieve near-optimal flow completion times [7, 57, 58, 71]. Additionally, storing the remaining flow size inside the packets helps our deflection component avoid deflecting and dropping packets of small flows that are more vulnerable to longer round trip times.[6] To realize SRPT marking, Vertigo receives flow size information from the application and keeps track of the remaining bytes of outgoing flows inside a hash table. Vertigo tags each packet with the remaining bytes of its flow.

Figure 3 depicts two potential implementations for Vertigo's auxiliary *flowinfo* header. A 32-bit field shows the flow's remaining bytes. Its uniqueness across packets of a single flow allows the ordering component to detect and resolve packet re-ordering. Additionally, we add a 4-bit counter, *retcnt*, to track the number of re-transmissions each packet has experienced, a 3-bit counter, *Flow-id*, to ensure correct ordering between subsequent flows, and a single-bit flag to mark the first packet of the flow (§3.1.2).

3.1.2 Detecting duplicate packets. Packet loss and re-transmissions are inevitable in lossy networks. To ensure that the *flowinfo* header information remains consistent, the marking component must detect duplicate packets and retrieve their original RFS field. Moreover, persistently deflecting or dropping packets of large flows may eventually lead to their starvation. To prevent this, Vertigo employs a *boosting mechanism* as part of the marking component in which the priority of re-transmitted packets is elevated by reducing their RFS values. Concretely, every time a packet is re-transmitted, Vertigo divides its RFS by a user-defined *boosting factor*. In our evaluations, unless stated otherwise, we set the boosting factor to 2. The boosting procedure diminishes the risk of starvation and repeated re-transmission timeouts, even for packets of large flows.

Vertigo's marking component detects re-transmissions by calculating a CRC hash of packet headers and looking it up in a cuckoo filter [27] to enable fast look-up and updates in the dataplane. It then applies the boosting function to the original RFS field of the packet stored in the flow table and increments the *retcnt* field in the *flowinfo* header. This operation needs to be reversible at the receiver to maintain the synchrony between the sending and receiving components; therefore, we confine the marking component to perform only bitwise rotations on the RFS field. Doing so relieves the network components from performing any computations on *flowinfo* header fields and allows the destination to apply an inverse boosting function and retrieve the original RFS of the packet. This limitation implies that the boosting factors must be chosen from powers of two, yet we show that even 2× boosting is enough to properly mitigate starvation.

With the boosting factor of 2, Vertigo sets the RFS for a re-transmitted packet to half of its previous value by performing a bitwise right rotation on the original RFS. This reduces the probability of it being selected again for deflection or dropping in the network. At the destination, Vertigo performs bitwise left rotations on the packet to retrieve the packet's original RFS. Using a 32-bit RFS field, Vertigo can support up to 16 re-transmissions for every packet. In §4.3, we evaluate the impact of boosting function on Vertigo's performance and illustrate that, while boosting can increase the percentage of completed queries by up to 60%, increasing the boosting factor has negligible impact on overall performance.

3.2 Selective Deflection in the Network

We assume that switches use output queues sorted by packet ranks, where *ranks* are the RFS fields provided by Vertigo's marking component.[7] We also assume that switch forwarding tables are pre-populated with the information of the next-hops for every destination. Inspired by the "power of two choices" paradigm [55], when a switch receives a packet that has more than one possible next hop, the switch compares the queue lengths of two random ports and sends the packet to the least loaded one. In addition to the simplicity of its hardware implementation, the power-of-two choices forwarding enables fine-grained traffic distribution, hence, delivering higher throughput guarantees. To apply SRPT scheduling, Vertigo sorts packets in each queue in increasing order of their RFS fields. When the link is idle and there are packets stored in the output queue, Vertigo transmits the packet with the smallest RFS, *i.e.,* the packet stored at the head of the queue.

To manage microbursts, Vertigo tries to exploit the networks' extra capacity to absorb microbursts. Concretely, when a switch receives a packet and its output queue is full, the switch selects the packet with larger RFS between this packet and the tail of its queue to deflect. For deflecting a packet, the switch randomly selects two queues and enqueues the packet into the least loaded one. If both queues are full, the switch enqueues the packet in one of them (randomly selected). This will result in one or more packets being dropped from the tail of this queue.

[6]We also evaluate Vertigo's performance while using flow-aging to mark the packets (§4.3). Vertigo achieves 30% higher mean QCT under 90% load while applying flow-aging discipline instead of SRPT.

[7]Recent works provide practical implementations of this abstraction [67, 69].

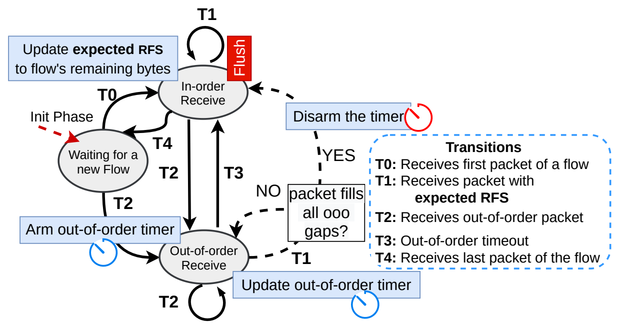

T1
Update expected RFS to flow's remaining bytes
Flush
In-order Receive
T0
Init Phase
Waiting for a new Flow
T4
T2
T3
T2
Arm out-of-order timer
Disarm the timer
YES
NO
packet fills all ooo gaps?
Transitions
T0: Receives first packet of a flow
T1: Receives packet with expected RFS
T2: Receives out-of-order packet
T3: Out-of-order timeout
T4: Receives last packet of the flow
Out-of-order Receive
T1
Update out-of-order timer
T2

Figure 4: The state-machine decides the fate of arrived packets. Out-of-order packets are placed inside a separate buffer, arming a timer to wait for delayed packets. In-order packets are immediately flushed to the network stack.

For both forwarding and deflecting packets, Vertigo uses the power-of-two choices paradigm to evenly distribute the load across the network. However, in contrast to forwarding (where a full queue results in *deflecting* packets), Vertigo *drops* packets if it encounters full buffers when it tries to deflect a packet. This distinction is intentional. While encountering a full buffer on the forward path can be a result of a local, short-lived microburst (*e.g.,* at the destination ToR in incast) and not necessarily an overall congested network, the output queue and two randomly selected ones all being full at the same time is a strong indication of extensive, extreme congestion. Via dropping packets in the latter case, Vertigo triggers the congestion control algorithms to throttle the send rates. Overall, Vertigo prioritizes packets of flows that are more likely to contribute to lasting congestion, *i.e.,* those with more remaining packets to transmit, for deflection and dropping. This helps Vertigo absorb microbursts even when the network is congested.

3.3 RX Path: The Ordering Component

The ordering component on the receive-side is the first software entity that obtains packets from the NIC. Its task is to detect out-of-order packets and temporarily buffer them until the delayed segments arrive or are timed out. To this end, Vertigo first extracts the *flowinfo* header. For every active flow, Vertigo stores the expected RFS and two intermediate linked lists containing references to the ready and out-of-order packets, respectively. RFS fields inside the *flowinfo* header are unique among the packets of a flow except for the re-transmitted packets. Since the re-transmission boosting mechanism may modify the RFS field (see §3.1.2), Vertigo first reverts the boosted RFS by applying *retcnt* left rotations on a packet's RFS field.

3.3.1 Ordered Receive. When the destination host receives a packet, it checks the RFS present in the *flowinfo* header. Vertigo expects a packet flagged as the initial packet of a flow to arrive first. Any packet with a larger counter indicates that there has been reordering or drops in the network. Figure 4 presents the state machine of the ordering component. After receiving the first packet of a flow, the state machine transitions to the *In-order Receive* state and continues to receive packets until an out-of-order packet arrives or the flow terminates. While in the *In-order Receive* state, arrived packets are placed in an intermediate buffer, and the protocol stack's receive routines are signaled to begin processing them immediately. The expected RFS is then updated by subtracting the received packet's size from the previously expected RFS value. However, when an out-of-order packet arrives, the ordering component designates a separate pre-allocated buffer to store early packets and wait temporarily for the in-transit packets. Upon receiving the entire flow, the state machine transitions back to the initial state (*Waiting for a new flow*). This is conceptually similar to the design of [31], except that Vertigo does not rely on GRO buffers or TCP sequence numbers to perform re-shuffling.

3.3.2 Out-of-order Receive. When in the *Out-of-order Receive* state, a timer is armed to prevent long pauses in packet processing caused by dropped packets. The waiting time depends on the network topology, link bandwidth, and link utilization. We define the waiting time (τ) as the maximum duration of time that the ordering component waits for a single delayed packet before starting to process out-of-order packets. While a short timeout increases the degree of reordering in the transport layer, large timeouts increase the flow completion tail latency. A safe estimate of τ is the time it takes for a single packet to traverse a network with almost full buffers. In the topologies we used in our evaluations, we set $\tau = 360\mu s$. §4.3 explores the impact of τ configuration on flow completion times.

Four events can occur when in *Out-of-order Receive*: 1) The host continues to receive early packets. In this case, the ordering component continues to buffer these packets along with their arrival timestamp until either timeout occurs or the expected packets arrive. 2) The host receives a packet that fills one of the gaps in the ordering buffer. Henceforth, the component can move its expected receive window forward to the next gap in the out-of-order buffer, update the timeout timer by subtracting the elapsed time from the arrival of next out-of-order packet from τ, and place the in-order packets in the *ready* buffer. After filling all the gaps, the state machine transitions back to *In-order Receive* state and disarms the timer. 3) The host receives packets with an RFS that is smaller than the expected RFS. This may indicate the arrival of delayed re-transmissions or a duplicate packet. Vertigo ignores packets that are already inside the ready buffer and places the late packets at the head of the ready buffer to send them up to the transport immediately. 4) The re-ordering timeout occurs. In this case, the re-ordering component releases all packets until the next gap up to the transport layer to trigger the protocol-specific decisions. It moves its expected packet pointer to the next gap in the out-of-order buffer and updates the timer. Finally, unless the out-of-order buffer is not empty, the system transitions to *In-order Receive*.

Previous proposals on random deflection disable TCP fast re-transmit to prevent the consequences of excessive re-ordering [65, 75], such as injection of redundant duplicate packets. This causes the packet loss to be detected only by re-transmission timeouts (RTOs), further delaying the loss recovery process. In Vertigo, we configure ordering timeouts to be small enough to still trigger fast re-transmission in the case of packet loss, while preventing the injection of unnecessary duplicate packets into the network. This ensures that VertigoâĂŹs interference with existing congestion control mechanisms is minimized.

Summary: We presented the design of Vertigo, a burst-tolerant routing technique for datacenters that deflects excess packets to neighboring switches to absorb microbursts. To prevent the performance degradation of deflection under high load, Vertigo prioritizes deflecting (and dropping, when the network is overloaded) the packets of the flows with more remaining bytes to send as they are more likely to contribute to lasting congestion. Furthermore, it employs a boosting mechanism to further assist with flow completions and prevent starvation. To enable these, we augmented the hosts' network stack with a transport-independent fast packet processing framework that performs packet marking on the TX path and packet ordering on the RX path.

4 PERFORMANCE EVALUATION

We evaluate Vertigo using large-scale OMNET++ simulations [2] and microbenchmarks on a physical testbed using our prototype host component implementation.[8] We briefly summarize our findings:

- Our large-scale simulations demonstrate that, using the DCTCP congestion control under 85% offered load, Vertigo reduces the average incast query completion times by 66%, 65%, and 77% compared to ECMP, DRILL, and DIBS, respectively (§4.2).
- We show that combining Vertigo with Swift's state-of-the-art congestion control can significantly reduce the packet loss under bursty traffic (85% load) and achieve 32×, and 10× improvement over Swift+DRILL in query completion times and flow completion times, respectively (§4.2).
- We analyze the impact of each individual component of Vertigo and show how the combination of selective deflection, SRPT forwarding, re-transmission boosting, and receiver-side ordering achieves strong burst-tolerance (§4.3).

4.1 Simulation Setup

Network Topologies. We perform our simulations on a two-tiered leaf-spine topology consisting of 4 core switches, 8 aggregate switches, and 320 servers. The switch links have 40Gbps bandwidth and the servers are connected to ToRs using 10Gbps links. Each switch port has 300KB buffer capacity [12, 75]. We also validate our findings against an equivalent fat-tree [4] network ($k = 8$) with 128 servers and 80 switches.

Workloads. For the background load, we run widely deployed, public datacenter traffic traces, *i.e.,* Facebook's cache follower, Facebook's data mining, and Google's web search, for interarrival times and flow size distributions [6, 62]. We scale the flow interarrival times to vary the load. For generating microbursts, we implement an incast application in which some randomly selected clients periodically send queries to a set of servers, also selected at random, that all reply to the queries simultaneously. To vary the level of traffic burstiness, we change three parameters: (a) the incast *scale*, *i.e.,* the number of servers that each client sends a query to, (b) the number of incast queries per second (QPS), *i.e.,* the rate at which the destinations initiate incast queries and (c) the individual incast flow size. We set the simulation time limit of the experiments to 5 seconds.

Alternative approaches. We compare Vertigo against in-network solutions like ECMP, the default load-balancing scheme widely used in datacenters, DRILL [33], a micro load-balancer with per-packet load distribution decisions, and DIBS [75], a random packet deflection technique. For transport and congestion control, we combine the above alternatives with TCP Reno [28], DCTCP [6], and Swift [47]. We also evaluate Vertigo's performance with alternative packet marking and scheduling disciplines.

[8]Vertigo artifacts for large-scale simulations, host implementation, and switch scheduler abstraction are publicly available at https://github.com/hopnets/vertigo-artifacts.

Setting	Min	Max	Default
Background Load % [48, 54, 75]	15	95	**50**
Incast QPS [75]	2000	28000	**4000**
Incast Scale [49, 75]	50	450	**100**
Incast flow size (KB) [75]	1	180	**40**
Reordering timeout (μs)	120	1080	**360**

Table 1: Parameter ranges used in Vertigo evaluation.

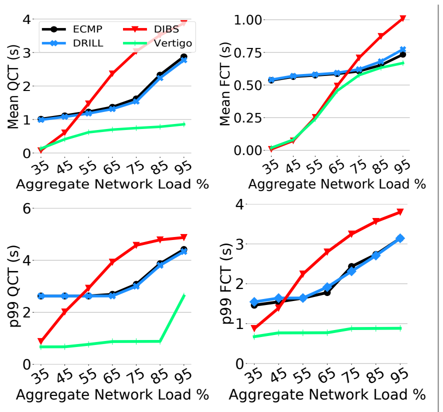
Parameter settings. Table 1 lists the parameters for large-scale simulations. Unless explicitly stated otherwise, we use the default values for all experiments, run DCTCP with the marking threshold of 65 as our default transport protocol, set Vertigo's deflection and load-balancing factors (power of n) to 2, TCP's initial window to 10 packets, and TCP's initial RTO to 1s and minRTO to 10ms to closely follow the parameter settings reported in [6, 75]. Lastly, we configure Swift according to the guidelines provided in [47]. All other parameters are default INET settings [1].

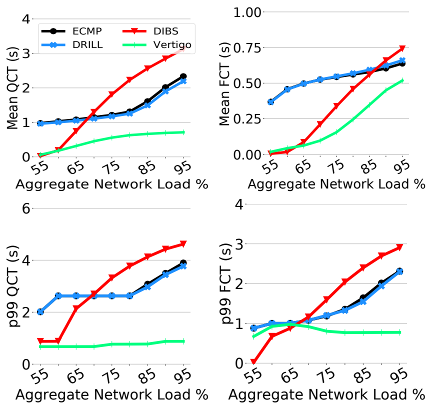
4.2 Large-scale Event-driven Simulations

We evaluate Vertigo under various incast and non-incast traffic settings by comparing it to widely deployed and the state-of-the-art techniques. We collect and report the response times, flow completion times (FCT), query completion times (QCT), application goodput, and some lower-level metrics like packet drops, reordering, and RTTs.

Vertigo offers superior performance under various degrees of load. In the first experiment, we gradually increase the incast event rate under three different degrees of background load, 25%, 50%, and 75%. The results, presented in Figure 5, clearly show the limitations of micro load balancing and random deflection. DRILL (a micro load balancer) cannot prevent last-hop bursts that are induced by incast. In practice, the majority of drops occur at the last hops [48, 68]. The performance of DIBS rapidly degrades under load, *e.g.,* with a 10% increase in the overall load from 60% (50% background+10% bursty workload) to 70% (50% background+20% bursty workload), the mean QCT and FCT of DIBS increase 6-fold and 21-fold, respectively. In comparison, the performance of other techniques degrades much more gracefully under load, *e.g.,* the QCT and FCT of DRILL increase by 10% and 16%, respectively. Overall, neither micro load balancing nor deflection alone is effective under both low and high loads. Vertigo, in contrast, consistently delivers low FCT and QCT under all degrees of load including extreme loads, *e.g.,* under 90% load (75% background + 15% incast load), Vertigo reduces the mean FCT of DRILL and DIBS by 5.1× and 2.7×, respectively. The results also show that when the background load dominates the incast load, DIBS's 99th FCT percentile (p99) remains lower than other systems due to saving persistent background flows, however, this performance comes at the cost of higher incast query completion times compared to Vertigo.

We repeat these experiments, replacing DCTCP with TCP for all schemes. Our results highlight the dependence of DIBS on DCTCP.


(a) 25% BG Load


(b) 50% BG Load

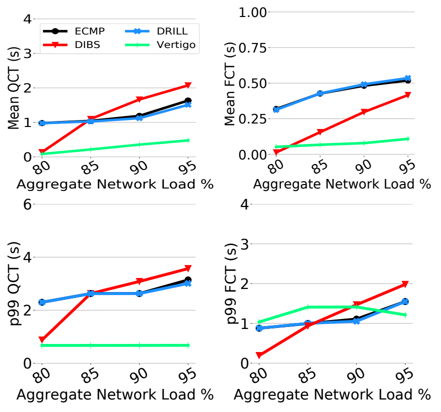
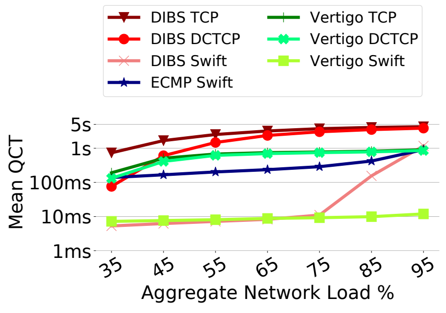

(c) 75% BG Load

Figure 5: Vertigo achieves a steady QCT performance under various load distributions when all systems use DCTCP as their transport. As the background load dominates the network, fewer packet drops lead to shorter query completion times.


(a) Mean QCT

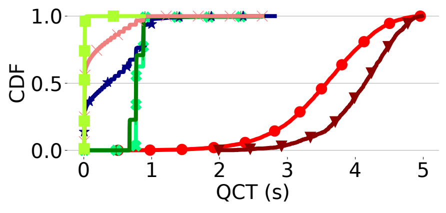

(b) QCT CDF

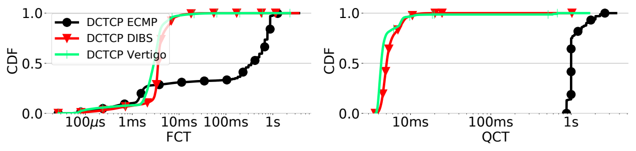
Figure 6: Vertigo delivers low QCT with TCP, DCTCP, and Swift.

Replacing DCTCP with TCP leads up to 10× jump in DIBS' query completion times and expedites its collapse under load. In contrast, Vertigo remains efficient under TCP as well. Figure 6 shows that Vertigo+TCP outperforms other alternatives that leverage DCTCP, and performs in the close proximity of Vertigo+DCTCP.

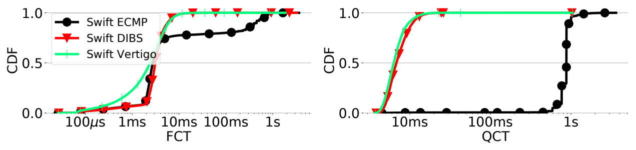
We next replace TCP with a state-of-the-art congestion control technique, Swift [47]. Designed specifically to handle extreme and bursty datacenter workloads such as large-scale incasts, Swift is different from traditional congestion control protocols such as DCTCP and TCP along two key dimensions: (1) it leverages advanced, high-resolution hardware and software timestamps to precisely measure RTTs and rapidly react to increasing RTTs and (2) it combines window-based congestion management with packet pacing to prevent microbursts. Under extremely large incasts, with thousands of flows destined to a single host simultaneously, the number of flows exceeds the path BDP (bandwidth-delay product) and even a congestion window of one single packet is too high to prevent packet drops. Window-based congestion control algorithms such as DCTCP and TCP are fundamentally not suited for managing such extreme microbursts. To handle such cases, Swift allows the congestion window to fall below one packet, *e.g.*, *cwnd*=0.5 results in sending a packet after a delay of 2RTT [47].

Consistent with the reports from Google's datacenters that show Swift's efficiency even under extreme load (*e.g.*, 95%) [47], our simulations show that Swift retains low-latency under different load levels and large-scale incasts. However, our results also show that the performance of Swift can be significantly improved if it is combined with Vertigo. By selectively deflecting packets, Vertigo delays pacing and rate reduction in Swift. For example, under 25% fixed background load combined with various levels of incast traffic raising the load up to 95%, running Vertigo with Swift results in an order of magnitude reduction in Swift's mean QCT and zero packet loss up until the load reaches 75%. Under 85% and 95% load, Vertigo+Swift drops only 2×10^{-6}% and 2×10^{-4}% of the packets, respectively. In comparison, with ECMP+Swift, the loss rates are 0.12% and 0.51%, respectively. Under Vertigo+Swift, the drop rates are up to four orders of magnitude lower than the drop rates under Vertigo+TCP and Vertigo+DCTCP. This underscores the importance of congestion control in managing severe congestion. Similar to our previous results, DIBS is effective under low load but fails as the load increases. Figure 6a depicts the results.

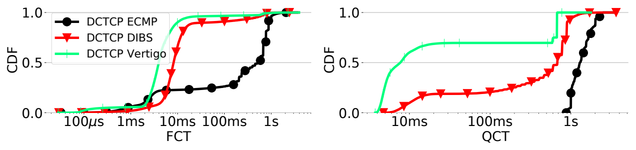
Vertigo is effective in three-tiered topologies. To evaluate Vertigo in another common datacenter topology, we perform our simulations in a fat-tree [4] with $k = 8$ (128 servers and 80 switches) and 10Gbps links, shortening the simulation deadline to three seconds. For these experiments, we compare ECMP, DIBS, and Vertigo under three different combinations of background and incast load when using DCTCP and Swift for congestion control. Based on the results shown in Figure 7, for a network filled with 50% background and 25% incast load, Vertigo can effectively reduce the QCT of ECMP under both DCTCP and Swift by 71% and 98%, respectively,



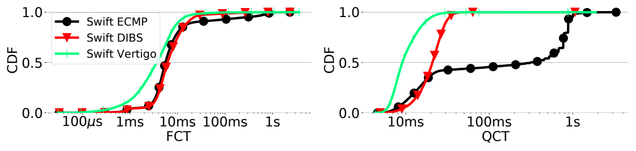
(a) 25% background + 10% incast - DCTCP



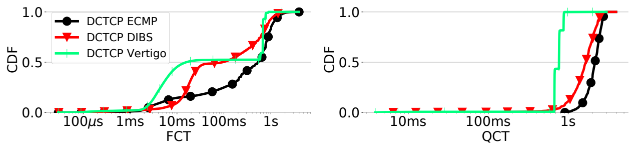
(b) 25% background + 10% incast - Swift



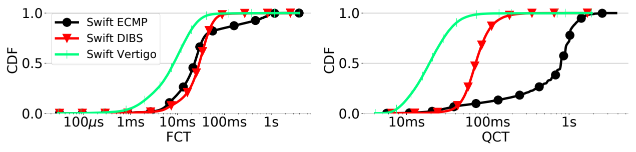
(c) 50% background + 25% incast - DCTCP



(d) 50% background + 25% incast - Swift



(e) 25% background + 60% incast - DCTCP



(f) 25% background + 60% incast - Swift

Figure 7: Vertigo cuts the flow and query completion times of ECMP and DIBS under two congestion control schemes in a fat-tree topology.

while improving the tail QCT of random deflection by 51% and 29%. This is because, with Swift, all systems experience orders of magnitude fewer drops. Therefore, even random deflection is able to considerably reduce the QCT tail of ECMP, and complete over 99% of the incast queries within the simulation deadline. Table 2 presents the percentage of all flows and incast queries completed before the deadline under both DCTCP and Swift. Our results indicate that, while ECMP and DRILL can heavily benefit from Swift, Vertigo is able to maintain over 98% and 93% flow and query completion performance, respectively, regardless of the congestion control technique.

Additionally, with less dominant background load, we observe that the QCT and FCT of DIBS quickly degrades with the increased incast traffic load, completing only 19% of the queries with DCTCP

CC/System	ECMP	DIBS	Vertigo
DCTCP	78.53%	96.07%	98.00%
Swift	97.69%	99.44%	99.76%

(a) % Flow completion

CC/System	ECMP	DIBS	Vertigo
DCTCP	28.36%	71.25%	92.98%
Swift	79.91%	99.06%	99.57%

(b) % Query completion

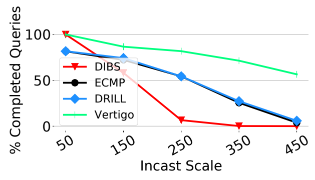
Table 2: Flow and query completion ratios of three evaluated systems under DCTCP and Swift congestion control under 75% load (50% background + 25% incast).

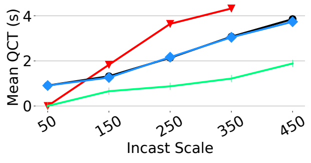
under 85% load. With 85% aggregate load consisting of 25% background and 60% incast (Figure 7f) and DCTCP as the transport protocol, Vertigo makes use of extra 33% buffering capacity, extra 20% deflection destinations, and extra 4× forwarding choices in the fat-tree topology to improve the percentage of completed queries and tail QCT of random deflection by 53% and 65%, respectively. Finally, we observe that in all scenarios Vertigo+Swift combination offers near-0 drops akin to the two-tiered leaf-spine simulations.

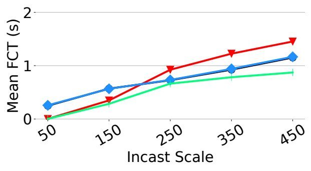
Vertigo completes more queries under large-scale incast. Next, we gradually increase the scale of incast events from 50 to 450 while fixing the incast rate to 4000QPS and incast flow size to 40KB. With 50% of the offered load originating from background traffic, the incast traffic elevates the overall offered load up to 95%. Figure 8 presents the results. As incast scale increases, all systems except Vertigo struggle to complete queries. That is because completing a single query requires the completion of many more individual flows. In contrast, Vertigo is able to selectively deflect, drop, and boost packets, resulting in up to 10× more completed queries compared to other alternatives. As Figure 8c shows, the mean FCTs of all schemes climb with higher incast scales which is a result of more frequent drops and queueing delays for both incast and background flows.

Vertigo performs well even under large incast flows. In this experiment, affixing the background load to 50%, we increase the overall offered load by sending larger incast flows. With 100 flows contributing to an incast event and incast rate of 4000QPS, we increase the size of the incast flows from 1K to 180KB. The results, presented in Figure 9, align well with our previous findings. As the incast flow size increases, the techniques that do not take remaining flow size into account fail to categorize large flows as incast flows and forward them accordingly. Vertigo, however, is able to identify halfway completed flows and helps them finish with the combination of deflection and re-transmission boosting mechanisms. As the flow sizes increase, DIBS's drop rate exceeds Vertigo. With 180KB incast flows, the mean QCT of Vertigo is 68% and 58% lower than DIBS and DCTCP+ECMP in the rightmost datapoints, respectively.

Burstiness is a challenge for random deflection. Vertigo performs robustly under extremely bursty traffic. Next, to investigate whether different degrees of burstiness can affect the overall performance of Vertigo, we fix the overall offered load to 25%, 50%, and 80%, adjusting the ratio of the incast traffic by squeezing the incast interarrival times. Figure 10 presents the results for 80% offered load. The QCT for all systems start to rise as the degree of burstiness increases. Vertigo, however, delivers steadily low latency by avoiding around 15% of packet drops and inflicting the majority of inevitable packet loss to larger flows even in the most extreme case. The results also show DIBS's limitation under heavy background load. With switch buffers partly occupied by


(a) Completed Incast Queries


(b) Mean QCT


(c) Mean FCT

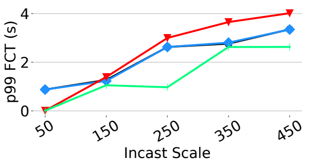

(d) p99 FCT

Figure 8: Vertigo completes up to 10× more queries in larger scale incast traffic patterns.

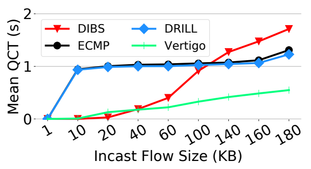


Figure 9: Vertigo maintains a high query completion rate even with large incast flows while other alternatives face frequent RTOs due to congested buffers.

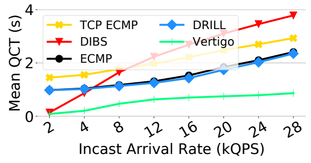


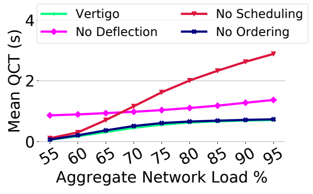
Figure 10: Tweaking the degree of burstiness by increasing the incast arrivals and affixing the overall offered load. Vertigo maintains low QCT performance.

background flow, DIBS quickly fails to handle incast queries as shown in Figure 10. Even under a low load (25%), we observe that Vertigo's mean QCT is 16% lower than DIBS. Under 25% and 50% load and extreme burstiness, Vertigo achieves 12% and 55% lower 99-%ile FCTs compared to DIBS, respectively.

Vertigo favors short flows under less bursty workloads. Incasts and microbursts are the norm in today's datacenter workloads [15, 44, 62, 68, 76]. However, we evaluate Vertigo under non-bursty traffic as well. Our results show that Vertigo continues to deliver a solid performance under these traffic patterns, too. We increase the background traffic, sampled from Facebook's cache follower, Facebook's data mining, and Google's Web search distributions, from 25% to 90% [6, 62]. Cache follower is a mice-dominated workload with 50% of the flows sending less than 24KB. Hence, Vertigo's SRPT forwarding combined with its micro load-balancing helps reduce the overall FCTs by up to 116% compared to ECMP+DCTCP. For web search and data mining workloads that are dominated by large flows, we observe that using Vertigo leads up to a marginal 4% increase in FCT. Our results indicate that without bursty traffic and for workloads dominated by large flows, transport protocols like DCTCP are effective in recovering from infrequent drops.

4.3 Vertigo Design Deep-dive

Component Analysis. Vertigo relies on three main components responsible for (1) deflecting packets that arrive at a full buffer (instead of simply dropping them), (2) SRPT scheduling in the network core (as opposed to FIFO), and (3) retrieving the correct ordering of out-of-order packets before sending them to higher layers. We quantify the impact of each of these components on the overall superior performance of Vertigo in Figure 11a. The key contribution of packet deflection is avoiding drops, and consequently, RTOs. We observe that in the lowest load, without deflection, QCT increases by a factor of 13× as the packet loss raises by 6×. Disabling the scheduler, however, has an even more notable effect as it reduces Vertigo's performance to that of random deflection alternatives. For example, in high loads, without scheduling, Vertigo suffers from up to 110% increase in mean QCT. We also repeat the experiments with Vertigo without its ordering layer. While packet re-ordering has minimal impact on QCT, it increases the overall FCT by up to 9% and reduces the overall goodput by 7% as reordering causes the larger flows' windows to shrink.


(a) Deflection and ordering

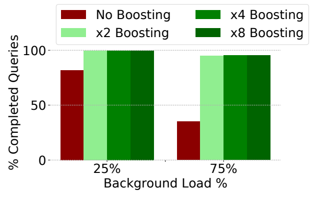
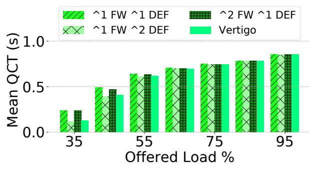

(b) Boosting

Figure 11: Each individual aspect of Vertigo's design (deflection, scheduling, ordering, and boosting) contributes to its efficiency.

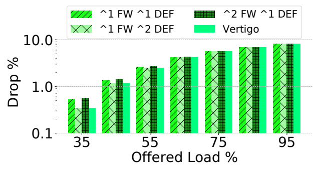
Re-transmission boosting is key in completing incast queries. Boosting the re-transmitted packets is another critical player in the overall performance of Vertigo. We start by disabling the boosting mechanism (the leftmost columns in Figure 11b). Then, starting from 2× (the default factor for reducing the packet RFS values for every re-transmission), we try various powers of two boosting factors up to 8×. We observe that boosting is essential in completing flows that experience packet drops; Vertigo's query completion rate drops by 65% without boosting. However, boosting packets more aggressively (boosting factors above 2×) does not cause a noticeable improvement, since the majority of re-transmitted packets successfully reach their destination in the second attempt with the boosting factor of 2×.

Comparing random and power-of-2 load-balancing on deflection and forwarding. Vertigo uses the power of two choices scheme for its forwarding and deflection decisions. To measure the impact of this load balancing paradigm on Vertigo's performance, we preform four sets of incast simulations with 50% background load and various incast QPS rates, trying out random and power of two choices load-balancing for forwarding (`1FW` and `2FW`, respectively) and deflection (`1DEF` and `2DEF`, respectively). We present our results in Figure 12. We observe that randomly choosing a destination for deflected packets increases the chance of packet drops by up to 47%, severely damaging the query completions. The gap, however, begins to fade as the offered load increases. This is because, in higher network loads, the probability of finding free buffer space is significantly reduced.

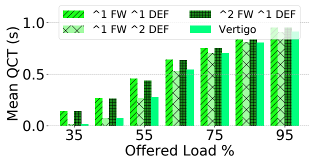
Alternative marking disciplines. Flow size information is a reliable indicator of flow persistence and can be used to achieve near-optimal scheduling performance. However, if the flow size



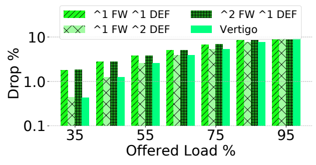
(a) QCT under two-tiered LS



(b) Drops under two-tiered LS



(c) QCT under fat-tree



(d) Drops under fat-tree

Figure 12: Evaluating random (⌒1) and power-of-two (⌒2) techniques for forwarding (FW) and deflection (DEF) in Vertigo. The power-of-two-choices paradigm reduces packet drops and QCTs under low and medium loads.

Setting → Load ↓	DCTCP ECMP	DCTCP DIBS	Vertigo SRPT	Vertigo LAS
55%	0.98	0.02	0.06	0.14
65%	1.08	0.74	0.32	0.40
75%	1.22	1.81	0.56	0.59
85%	1.61	2.56	0.67	0.77
95%	2.34	3.12	0.72	0.94

Table 3: Flow aging is less effective than SRPT, but outperforms other baselines.

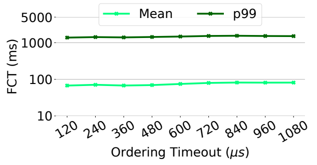


Figure 13: Ordering timeout configuration has a negligible impact on flow completion times.

information is not available, Vertigo resorts to *flow aging* instead. Under this scheduling discipline, also known as *Least Attained Service (LAS)*, flows are initially marked with 0, regardless of their size. The consecutive packets carry a counter that shows the total number of packets sent by their flow or flow's age in packet units. We adapted the host components to implement LAS and repeated our previous simulations. Table 3 shows the results. LAS takes a few transmissions to differentiate between flows. Consequently, its initial decisions are suboptimal compared to SRPT. We observed up to 30% difference in the mean QCTs under SRPT and LAS with different workloads. However, Vertigo+LAS still outperforms other baselines such as ECMP and DIBS by 52% and 70% under 85% offered load, respectively.

Vertigo's re-ordering timeout setting has a bounded effect on flow completion times. Finally, to demonstrate the effectiveness of Vertigo's ordering extension in detecting and resolving packet re-ordering, we repeat our incast simulations, raising the ordering timeout (τ) from $120\mu s$ up to 1.08ms. The results presented in Figure 13 suggest that setting the timeout to the maximum delay a packet might experience in the network without being deflected, as described in §3.3, is indeed a good estimate as the majority of out of order packets are not deflected. We observe that the latency penalty of improper timeout setting does not exceed a few milliseconds (5ms in our simulations) in abundance of incast flows in an extremely congested network.

4.4 End-host and Switch Implementation

We deploy Vertigo's host components, on a physical testbed consisting of two Cloudlab [25] machines, featuring two Intel Xeon E5-2640v4 processors, 64GB of memory, and 25Gbps Mellanox ConnectX-4 network interface cards. We build a packet generator tool on top of Vertigo's userspace network stack and measure the RTTs and throughput of a single-threaded TCP server with the marking and ordering components enabled/disabled. To ensure minimal packet processing overheads in Vertigo, we use DPDK cuckoo filters for flow identification at both the marking and ordering components. For the packet header hash, we use pre-allocated pools of fast hash tables and modify the existing ring buffer implementations in DPDK to store out-of-order packets. Our results indicate that the two extra hash table look-ups performed by Vertigo's marking component require an additional 300ns processing time. However, the throughput results demonstrate <0.1% difference when Vertigo's marking component is enabled.

Recent endeavors on designing hardware-based priority queues enable programmable scheduling in data-plane at line-rate [64, 67, 69]. To implement a Vertigo switch, two mechanisms must be added to existing implementations of the priority queue: 1. *The ability to extract items from the tail of the priority list.* In Vertigo, packets that arrive at a full buffer still might get a buffering space if their remaining flow size is smaller than that of a buffered packet. To this end, we extend PIEO's abstractions [67] to support packet extraction from the tail of the queue (§A.3). Similar to other functionalities supported by PIEO, the packet extraction process is executed in 4 cycles. 2. *The ability to change the output port for a packet when a buffer is full.* With *negative mirroring* functionality already available in PISA switches [3] and fast SRAM access in re-configurable switches [67], Vertigo can perform deflection in just a few clock cycles. To that end, we extended the functionality of PIEO [67] to find that deflection, a special case of enqueue operation requires one additional dequeue, adding 4 extra cycles. Since packet drops remain scant compared to the overall flowing traffic, the amortized cost of insertion remains unchanged. We leave the full hardware implementation of a Vertigo switch and testbed evaluations using this switch to future work.

5 RELATED WORK

Datacenter traffic is bursty [15, 44, 62, 76]. Microbursts have a large set of root causes such as bursty traffic introduced by datacenter applications [11, 32, 73], transport protocol operations [40, 41], and operating system optimizations [44, 53], that are not hard to detect and manage [42]. Some recent in-network designs augment the programmable data planes with fast stateful processing to detect microbursts. Conquest [20] introduces a queue monitoring technique to detect heavy-hitter flows in the presence of microbursts. Marple [59] presents a data store and a query language that can track and detect fine-grain traffic characteristics (*e.g.,* inter-packet gap patterns for a flow). Mantis [74] enables a datapath between the switch dataplane, to gather the traffic state, and switch control-plane to implement quick reactive decisions. Although these monitoring tools are invaluable for network management, we find that a simple host-assisted technique that tags packets with their flow size information is surprisingly effective for managing microbursts.

Preventing microbursts at the edge. Many recent proposals try to *prevent the formation of microbursts at end-hosts* [6, 29, 30, 47, 49, 56]. Imposing random delays to user requests to mitigate the degree of synchronization between flows [29] is among the first deployed remedies. This technique improves the tail of Flow Completion Time (FCT) but increases the median FCT [6]. Receiver-driven transports are another emerging solution to avoid packet loss by *performing bandwidth allocation using credit packets* [22, 30, 39, 56]. Another group of works try to *prevent the sender hosts from transmitting large bursts of traffic that the network cannot absorb* [6, 47, 48]. Using Explicit Congestion Notification (ECN), DCTCP tries to avoid microbursts' consequences by keeping headroom inside switch buffers to absorb the bursts [6]. Swift [47] and HPCC [48] improve the precision and speed of prior congestion control algorithms such as DCTCP via using advanced telemetry techniques. Swift uses fine-grained timestamps at the edge to accurately measure packets' RTTs as a congestion notification and paces the packets accordingly to prevent congestions. HPCC [48], an RDMA congestion control algorithm, relies on in-network telemetry information carried by in-flight packets to adjust its transmission rate and avoid packet loss. These techniques require a few Round-Trip Times (RTTs) to converge, which is longer than the lifetime of most microburst events [76]. Although significantly faster than TCP and DCTCP, we show that Swift can benefit from running Vertigo in the networking layer, *e.g.,* under 55% load, Swift+Vertigo's mean QCT is 96% shorter than Swift+ECMP (§4).

Taming microbursts in the network core. Datacenter load balancers [5, 33, 36, 72] strive to *distribute microbursts at the core* of the network by evenly balancing the traffic among multiple paths. These techniques fundamentally cannot manage the microbursts at the last hop (switch-to-receiver host) where the majority of microbursts transpire [48, 68, 76]. Unlike others, FastPass [61] uses a centralized arbiter to schedule and route the traffic. While FastPass can prevent microbursts, centralized designs pose scalability challenges. Deflection techniques detour excess packets to neighboring switches [65, 75]. Deflection, however, creates multiple challenges in datacenters. Notably, it results in excessive packet re-ordering, introduces head-of-the-line blocking in switch buffers, and breaks under high load (§2).

Buffer management, *e.g.,* via packet scheduling (ordering packets within a queue) and selective dropping (in case of buffer overflow), is a large and mature area of research [7, 16, 23, 24, 35, 37, 49, 60, 66]. NDP, for example, drops the packet payload [35] and TLT prevents packet drops that are only recoverable by timeouts [49]. We find that a simple buffer management technique based on SRPT and the remaining flow size is efficient across all the tested workloads. We leave a detailed exploration of optimizing Vertigo with other buffer management techniques to future work.

Some L2/L3 techniques rely on the internals of the congestion control algorithm to implement networking services [31, 36, 75]. DIBS, for instance, disables the fast retransmission mechanism of DCTCP [75], and Presto and Juggler use TCP's sequence numbers to resequence the reordered packets [31, 36]. This poses a hurdle in deploying such techniques as the set of congestion control algorithms in today's datacenters is large and rapidly evolves to meet key operational needs [47, 48, 53]. Plus, even in one datacenter, many congestion control algorithms can coexist, *e.g.,* latency-sensitive and WAN traffic deploy different congestion control algorithms, customers configure their cloud VMs with their preferred congestion control algorithms, and UDP traffic relies on the application-level rate control logic [47]. Requiring changes to the congestion control logic and relying on its internal mechanisms complicate deployment. Thus, Vertigo strives to provide a burst-tolerant forwarding service that is agnostic to the internals of the congestion control algorithm. This allows these layers to evolve independently. In Vertigo, we build on some of the powerful ideas from the related work, *e.g.,* "power of two choices" forwarding [33], deflection routing [75], and ordering layers [31], and address their limitations and shortcomings.

Integration with network monitoring. The continuous growth in the scale of datacenters and the rate at which they operate, in addition to higher performance expectations and more strict service-level objectives (SLOs), leave minimal time slack for resolving network anomalies [52, 77]. As finding the root causes of network anomalies requires considerable time, the proposals on network telemetry [14, 46, 52, 77] advocate fast, accurate, and scalable network monitoring. Concretely, to quickly pinpoint the cause of a network anomaly and SLO violation, these proposals seek fine-grained measurement of performance-critical characteristics of the network, such as link utilization, queue occupancy, path conformance, and packet drop. Vertigo's functionalities might interfere with some of these telemetry operations. For instance, tracking the number of packet drops gives the operators an insight into the degree of temporal congestion events inside the network while, with packet deflection, packet drops only indicate large-scale long-lasting congestion. However, by tracking other events, such as link utilization and the number of deflections per packet, a telemetry system can detect anomalies such as temporal congestion. We leave the design of a telemetry system that supports packet deflection to future work.

6 CONCLUSION

Short-lived congestion events that cause excessive packet loss remain a major challenge in datacenter networks. Frequently low utilization is datacenters suggests that temporarily deflecting burst packets to switches with spare capacity may prevent packet loss. In this study we identify the main issues that stem from random packet deflection and address them by introducing Vertigo, a hybrid design that incorporate end-hosts' knowledge of the workload and the network's immediate reaction capacity to selectively deflect packets that arrive at full buffers to the neighboring switches. Our simulation results demonstrate that Vertigo provides up to 3.3× lower mean incast query completion times and 3× lower 99-%ile flow completion times compared to DCTCP+ECMP while improving the mean QCT by 18× when combined with Swift.

ACKNOWLEDGMENTS

We thank our shepherd, Yikai Lin, and all the anonymous CoNEXT reviewers for their valuable feedback. We also thank Sougol Gheissi for her help and feedback during the course of the project. This project was partially supported by a Facebook faculty research award.

REFERENCES

[1] 2020. *INET Framework. https://inet.omnetpp.org/.*

[2] 2020. *OMNeT++ Simulator. https://omnetpp.org/.*

[3] 2020. *Open Tofino. https://github.com/barefootnetworks/Open-Tofino.*

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM.*

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: distributed congestion-aware load balancing for datacenters. In *SIGCOMM.*

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM.*

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: minimal near-optimal datacenter transport. In *SIGCOMM.*

[8] Mark Allman and Ethan Blanton. 2005. Notes on burst mitigation for transport protocols. *SIGCOMM CCR* (2005).

[9] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically finding the cause of packet drops. In *NSDI.*

[10] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *SIGCOMM.*

[11] Z. Abbasi G. Gibson B. Mueller J. Small J. Zelenka B. Welch, M. Unangst and B. Zhou. 2008. Scalable Performance of the Panasas Parallel File System. In *FAST.*

[12] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. 2016. Enabling ECN in multi-service multi-queue data centers. In *NSDI.*

[13] Neda Beheshti, Petr Lapukhov, and Yashar Ganjali. 2019. Buffer Sizing Experiments at Facebook. In *ACM BS.*

[14] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *SIGCOMM '20.*

[15] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *IMC.*

[16] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. 1998. An architecture for differentiated services. *RFC 2475* (1998).

[17] Alberto Bononi, Fabrizio Forghieri, and Paul R Prucnal. 1993. Analysis of one-buffer deflection routing in ultra-fast optical mesh networks. In *INFOCOM.*

[18] Flaminio Borgonovo, Luigi Fratta, and Joseph Bannister. 1993. Unslotted deflection routing in all-optical networks. In *GLOBECOM.*

[19] Flaminio Borgonovo, Luigi Fratta, and Joseph A Bannister. 1994. On the design of optical deflection-routing networks. In *INFOCOM.*

[20] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *CoNEXT.*

[21] Yang Chen, Hongyi Wu, Dahai Xu, and Chunming Qiao. 2003. Performance analysis of optical burst switched node with deflection routing. In *IEEE International Conference on Communications*, Vol. 2.

[22] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *SIGCOMM.*

[23] David D Clark, Scott Shenker, and Lixia Zhang. 1992. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *SIGCOMM CCR.*

[24] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. *SIGCOMM CCR* (1989).

[25] 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. 2019. The Design and Operation of CloudLab. In *ATC.*

[26] Chris Fallin, Greg Nazario, Xiangyao Yu, Kevin Chang, Rachata Ausavarungnirun, and Onur Mutlu. 2012. MinBD: Minimally-buffered deflection routing for energy-efficient interconnect. In *IEEE/ACM International Symposium on Networks-on-Chip.*

[27] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT.*

[28] Sally Floyd, Andrei Gurtov, and Tom Henderson. 2004. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782.

[29] S. Floyd and V. Jacobson. 1994. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking* (1994).

[30] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT.*

[31] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. 2016. Juggler: a practical reordering resilient network stack for datacenters. In *EuroSys.*

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP.*

[33] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *SIGCOMM.*

[34] Soroush Haeri and Ljiljana Trajković. 2014. Intelligent deflection routing in buffer-less networks. *IEEE Transactions on Cybernetics* 45, 2 (2014).

[35] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM.*

[36] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM.*

[37] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. In *SIGCOMM.*

[38] Ching-Fang Hsu, Te-Lung Liu, and Nen-Fu Huang. 2002. Performance analysis of deflection routing in optical burst-switched networks. In *Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1.

[39] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. 2020. Aeolus: A Building Block for Proactive Transport in Datacenters. In *SIGCOMM.*

[40] Hao Jiang and Constantinos Dovrolis. 2003. Source-Level IP Packet Bursts: Causes and Effects. In *IMC.*

[41] Hao Jiang and Constantinos Dovrolis. 2005. Why is the Internet Traffic Bursty in Short Time Scales? *SIGMETRICS Perform. Eval. Rev.* (2005).

[42] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks. In *APSys.*

[43] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The Nature of Data Center Traffic: Measurements & Analysis. In *IMC.*

[44] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. 2013. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *CoNEXT.*

[45] Kazuki Kawanabe and Tatsuro Takahashi. 2007. Effective deflection control method in optical packet switching networks with shared buffers. *Electronics and Communications in Japan (Part I: Communications)* 90, 9 (2007).

[46] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *SIGCOMM.*

[47] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M G Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *SIGCOMM.*

[48] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: high precision congestion control. In *SIGCOMM.*

[49] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *EuroSys.*

[50] Zhonghai Lu, Mingchen Zhong, and Axel Jantsch. 2006. Evaluation of on-chip networks using deflection routing. In *ACM Great Lakes symposium on VLSI.*

[51] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. 2006. Receive Side Coalescing for Accelerating TCP/IP Processing. In *HiPC.*

[52] Jonatas Marques, Kirill Levchenko, and Luciano Gaspary. 2020. IntSight: Diagnosing SLO Violations with in-Band Network Telemetry. In *CoNEXT.*

[53] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *SOSP.*

[54] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM.*

[55] Michael Mitzenmacher, AndrÃľa W. Richa, and Ramesh Sitaraman. 2000. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing.*

[56] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *SIGCOMM.*

[57] Ali Munir, Ghufran Baig, Syed M Irteza, Ihsan A Qazi, Alex X Liu, and Fahad R Dogar. 2014. Friends, not foes: synthesizing existing transport strategies for data center networks. In *SIGCOMM.*

[58] Aisha Mushtaq, Radhika Mittal, James McCauley, Mohammad Alizadeh, Sylvia Ratnasamy, and Scott Shenker. 2019. Datacenter congestion control: identifying what is essential and making it practical. *SIGCOMM CCR* (2019).

[59] S Narayana, A Sivaraman, V Nathan, P Goyal, and others. 2017. Language-directed hardware design for network performance monitoring. In *SIGCOMM*.

[60] Abhay K Parekh and Robert G Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking* (1993).

[61] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized" zero-queue" datacenter network. In *SIGCOMM*.

[62] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*.

[63] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. 2018. Micro-Burst in Data Centers: Observations, Analysis, and Mitigations. In *IEEE ICNP*.

[64] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable calendar queues for high-speed packet scheduling. In *NSDI*.

[65] X. Shi, L. Wang, F. Zhang, K. Zheng, and Z. Liu. 2017. PABO: Congestion mitigation via packet bounce. In *IEEE ICC*.

[66] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. *SIGCOMM CCR*.

[67] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *SIGCOMM*.

[68] 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. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*.

[69] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*.

[70] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. 2004. Dynamics of hot-potato routing in IP networks. In *International Conference on Measurement and Modeling of Computer Systems*.

[71] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption?. In *NSDI*.

[72] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI*.

[73] J Woodruff, A W Moore, and N Zilberman. 2019. Measuring Burstiness in Data Center Applications. In *BS*.

[74] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive Programmable Switches. In *SIGCOMM*.

[75] Kyriakos Zarifis, Rui Miao, Matt Calder, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. 2014. DIBS: just-in-time congestion mitigation for data centers. In *Eurosys*.

[76] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution measurement of data center microbursts. In *IMC*.

[77] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. 2020. Flow Event Telemetry on Programmable Data Plane. In *SIGCOMM*.

A VERTIGO ARTIFACTS

Vertigo artifacts are publicly available in our GitHub super-repository (https://github.com/hopnets/vertigo-artifacts). The super-repository points to three public submodules containing the code for Vertigo simulations, Vertigo host implementation, and Vertigo switch scheduler implementation. We briefly describe each module below and refer the readers to individual README files for a detailed guide on building and running experiments.

A.1 Omnet++ simulations

Available at (https://github.com/hopnets/vertigo_simulations). We use Ubuntu 18.04, Omnetpp-5.6.2 [2], and INET framework [1] to run our simulations. The following steps are required to run the simulations:

(1) Installing Omnet++ simulator
(2) Installing the project's dependencies
(3) Downloading and building the project modules
(4) Running the simulations and extracting the results

A detailed guide on running the simulations can be found in the README file.

A.2 Host Implementation

We used two Cloudlab [25] machines as described in §4.4 to deploy and evaluate host components: marking and ordering. The artifacts are available at (https://github.com/hopnets/vertigo_host). We recommend using Mellanox NICs with kernel-bypass support for the evaluation purposes. A detailed guide on building and evaluating the host components can be found in the README file.

A.3 Switch scheduler implementation

Finally, the repository (https://github.com/hopnets/vertigo_scheduler_fpga_implementation) contains Verilog source for implementing switch scheduler operations based on PIEO [67] for Intel FPGA devices. Please contact authors for further questions regarding the artifacts.