BurstBalancer: Do Less, Better Balance for Large-scale Data Center Traffic

Paper # 45

***Abstract*—Layer-3 load balancing is a key topic in the networking field. It is well acknowledged that flowlet is the most promising solution because of its good trade-off between load balance and packet reordering. However, we find its one significant limitation: it makes the forwarding paths of flows unpredictable. To address this limitation, this paper presents BurstBalancer, a simple yet efficient load balancing system with a sketch, named BalanceSketch. Our design philosophy is *doing less changes* to keep the forwarding path of most flows fixed, which guides the design of BalanceSketch and balance operations. We have fully implemented BurstBalancer in a small-scale testbed built with Tofino switches, and conducted large-scale NS-2 simulations. Our results show that BurstBalancer achieves 5%$\sim$35% and up to 30$\times$ smaller FCT in symmetric and asymmetric topologies, respectively, while 58$\times$ less flows suffer from path changing. All related codes are open-sourced anonymously[1].**

I. Introduction

A. Background and Motivation

To support the ever-increasing traffic demands of cloud services, the number of data centers increases rapidly [1]–[5]. Typical data center networks (DCNs) use symmetric topologies (*e.g.*, Fat-Tree [6], VL2 [7], DCell [8]) where many candidate paths exist between any server pair. How to evenly allocate the traffic to these candidate paths is well known as the layer-3 (L3) load balance. L3 load balance has been acknowledged as one key topic in the networking field for many years [9]–[15].

There are three types of L3 load balancing schemes. First, packet-level load balancing schemes [16]–[24] select a path for each packet and achieve perfect traffic split. However, they suffer from serious reordering problems when the delay of candidate paths has a large difference. Second, flow-level load balancing schemes [25]–[36] assign one path to all packets of each flow. They avoid packet reordering, but cannot well balance the traffic due to the skewed distribution of flow sizes and hash collisions among large flows [37]. Third, flowlet-level load balancing schemes [1], [38]–[43] make the trade-off between minimizing packet reordering and evenly balancing traffic. In their design, the packets of a flow are divided into many groups, where the time interval between any two adjacent groups is larger than a predefined threshold δ. Each group of packets is called a flowlet [38]. In other words, they divide every flow into many flowlets using δ. It is well acknowledged that flowlet is the most promising solution because of its good trade-off between packet reordering and load balance [2], [43]–[46]. However, they cannot precisely detect flowlets using small memory, and make a lot of unnecessary manipulation: 1) The forwarding paths of the flows are unfixed and unpredictable, while being aware of the paths is essential for network measurement and management. 2) Due to the limited memory on hardware and large concurrent flows, they inevitably regard multiple flows as one flow, leading the number of flowlets decreases a lot. 3) They unnecessarily divide small flows into many flowlets, increasing the risk of packet reordering. In summary, to better balance the load, all existing solutions manipulate many or all packets/flows in some ways. The more packets/flows manipulated, the more chaotic the network will be.

Although existing load balancing schemes have made excellent contributions, they do not consider the *flow-regulation* of the network. Flow-Regulation means that given a flow ID, its forwarding path can be easily calculated, and does not change with time. In most existing schemes, the forwarding path of a flow is unfixed and unpredictable, which brings great challenges for network management and optimization. Intuitively, if most members of a group follow a simple rule, then the management of this group would be simple. For many network operations, such as network diagnosis [47]–[52], congestion control [53]–[58], network measurement and management [48], [59]–[62], it is often assumed or expected that the forwarding paths of most flows can be obtained easily. For example, the well known 007 system [52] is designed for a network where all flows use ECMP. It needs the forwarding paths of flows to locate the congested link. If the forwarding path of flow changes rapidly and randomly, 007 cannot pinpoint the congested link timely and accurately, resulting in unreliable diagnose results. For another example, the pioneering work using INT for congestion control, HPCC [57], uses the link load information to adjust the sending rate of flows. If the forwarding paths of flows are fixed, HPCC works excellently; but otherwise, the link load information cannot match the culprit flow, so the advantages of HPCC cannot be guaranteed. Therefore, we want a solution that can not only balance the traffic well, but also keep the network traffic following the flow rule as much as possible. The ***ideal solution*** *should manipulate as few packets/flows as possible*, so as to make network measurement and management easier.

B. Our Proposed Solution

Towards the above goal, we propose BurstBalancer, an efficient load balancing system, with the aim of manipulating only a small number of critical flowlets, namely FlowBursts. In BurstBalancer, most flows follow ECMP [25] and we can

[1] https://github.com/BurstBalancer/Burst-Balancer

easily get their forwarding paths. BurstBalancer devises a sketch, namely BalanceSketch, and deploys it on each switch to detect and make forwarding decisions for each FlowBurst. Thanks to the memory efficiency of BalanceSketch, BurstBalancer only needs small on-chip memory to keep critical flowlets (FlowBursts), and thus perfectly embraces the highly skewed flow distribution [63]–[67]. Further, BurstBalancer only manipulates the critical flowlets which are very limited in number, minimizing packet reordering. In addition, BurstBalancer is easy to implement without any changes to end-hosts or protocol stacks, and can be incrementally deployed in existing networks.

The design philosophy of our BurstBalancer is ***doing less** manipulations while **better balancing** the traffic*, which is guided by the well-known Occam's Razor principle: entities should not be added beyond necessity. The philosophy of *doing less* includes two dimensions based on our two key observations. The first dimension of doing less is based on *Observation I*: only a minority of flowlets are fast and large enough to cause load imbalance, and we call these critical flowlets *FlowBursts*[2]. Therefore, we *manipulate only critical flowlets (FlowBursts).* For example, in our simulation experiments, there are about 18,000 concurrent flowlets, of which only 1.1% are *FlowBurst*. Therefore, if we identify, maintain, and manipulate only FlowBursts, it is possible to save on-chip memory up to 100 times while achieving similar load balance performance as those schemes identifying all flowlets. We believe that balancing only critical flowlets is a promising direction of L3 load balance. In this way, we divide all flowlets into two kinds: FlowBursts and unnecessary flowlets[3], and only manipulate FlowBursts. Detecting unnecessary flowlets requires huge memory overhead, and manipulating them only makes the network more chaotic and increases reordering.

The second dimension of doing less is based on *Observation II*: It is expensive and unnecessary to accurately detect and manipulate all FlowBursts. Therefore, we *only manipulate most rather than all FlowBursts.* 1) Finding all FlowBursts is expensive for current hardware resources. 2) Manipulating most FlowBursts while leaving other FlowBursts to follow ECMP path can achieve good balance. 3) Finding all FlowBursts needs complicated design of data structure. A strawman solution to identify FlowBursts is to first identify all flowlets using existing methods and then check whether the identified flowlet is a FlowBurst. However, this solution is memory inefficient because it records the information of all flowlets, most of which are unnecessary to manipulate. Therefore, we propose a simple data structure, namely BalanceSketch, to keep most rather than all FlowBursts (See details in § III) and evict unnecessary flowlets.

We extensively evaluate BurstBalancer on both small-scale testbed and large-scale simulation platforms. Our testbed consists of 4 Tofino switches [68] and 8 end-hosts in a leaf-spine topology. For simulations, we use an event-level simulator (NS-2 [69]) to test the performance of BurstBalancer in large-scale topologies. Our experimental results show that compared to LetFlow [1], BurstBalancer better balances the traffic using smaller memory, while manipulates 58× less flows at the same time. In symmetric topologies, BurstBalancer achieves 5%∼35% smaller FCT (flow completion time) than state-of-the-art LetFlow [1] and DRILL [16]. In asymmetric topologies, BurstBalancer achieves up to 30× smaller FCT than LetFlow and up to 6.4× smaller FCT than WCMP [26]. We also conduct CPU experiments, and results show that BurstBalancer achieves about 90% recall rate in finding FlowBursts with small memory. All related codes are open-sourced anonymously [70].

[2]A formal definition of FlowBurst is provided in § II-A.

[3]Here, we also give the definition of unnecessary flowlets: 1) flowlets formed by small flows; 2) flowlets formed by low-density flows (*e.g.*, some persistent flows that last for long time but send packets at a very slow speed).

TABLE I: Symbols frequently used in this paper.

Notation	Meaning
δ	Flowlet threshold that spaces two adjacent flowlets or FlowBursts
$\mathcal{V}$	Lower bound of the speed of FlowBurst
$\mathcal{F}$	Voting threshold used for identifying flowlets of high speed and large size
$\varDelta$	Flow timeout threshold used for identifying whether a flow ends
l	Number of buckets in BalanceSketch
$\mathcal{B}[i]$	The i^{th} bucket of BalanceSketch
$h(.)$	Hash function mapping each flow into one bucket in BalanceSketch

II. Background and Related Work

In this section, we begin with the problem statement of FlowBurst in § II-A. Then we discuss the related work of load balance solutions for data center networks in § II-B. The main symbols used in this paper are shown in Table I.

A. Problem Statement

Network Stream: A network stream is an unbounded timing evolving sequence of items $S = \{p_1, p_2, \cdots\}$, where each item $p_i = (f_i, t_i)$ indicates a packet of flow f_i arriving at time t_i.
Flow: A flow consists of packets $\{p'_1, \cdots, p'_n\}$ sharing the same flow ID f_i, which can be any combination of 5-tuple: source IP address, source port, destination IP address, destination port, protocol type.
Flowlet: Given a predefined flowlet threshold δ, a flowlet refers to a group of continuous packets $\{p'_1, \cdots, p'_m\}$ of a given flow f_i, such that $\forall 0 < j < m$, $t_{j+1} - t_j \leq \delta$. This flowlet is *active* if $|t_{now} - t_m| < \delta$, where t_{now} is the current time, and is *outdated* otherwise. Intuitively, *the packets of a flow are divided into many groups/flowlets, where the interval between flowlets is large enough.*
FlowBurst: For a flowlet $\{p'_1, \cdots, p'_m\}$, we define its size as m, and define its speed as $\frac{m}{\varDelta T}$, where $\varDelta T = t_m - t_1$. This flowlet is a FlowBurst if $\frac{m}{\varDelta T} > \mathcal{V}$ and $m > \eta_k$, where η_k is the size of the k^{th} largest flowlet among all active flowlets whose speed are larger than $\mathcal{V}$. Intuitively, *FlowBursts refer to a particular kind of flowlets that are fast and large enough to cause load imbalance.* For all active flowlets whose speed exceed a predefined threshold $\mathcal{V}$, we define the flowlets of the largest k sizes as the FlowBursts.

B. Related Work

Existing load balancing solutions for data centers can be roughly divided into three classes: packet-level schemes, flow-level schemes, and flowlet-level schemes. For other solutions, please refer to references [43], [71]–[74].

1) Packet-level schemes. Packet-level schemes [16]–[24] choose a desirable path for each packet. They achieve the ideal splitting ratio at the cost of packet reordering. DRILL [16] makes per-packet decisions at each switch based on local-queue occupancies and randomized algorithms. NDP [17] presents a multipath-aware transport-layer protocol that manipulates each packet, and introduces a handshake mechanism to alleviate reordering. MP-RDMA [18] proposes a per-packet multi-path protocol for RDMA network, where the packets are distributed in a congestion-aware manner. Other packet-level schemes include Fastpass [24], DeTail [23], QDAPS [19], RMC [20], OPER [21], and DRB [22].

2) Flow-level schemes. Flow-level load balancing schemes [25]–[36] assign a path to each flow. They avoid packet reordering but cannot well balance the traffic because of collisions between large flows. The well-known ECMP [25] uses flow-level hashing to select a path for each flow, and achieves excellent performance when there are only small flows but no large flows [34], [35]. WCMP [26] assigns each path a weighted cost, and distributes the traffic based on the cost. MPTCP [33] splits each TCP flow into several subflows, and assigns each subflow to a non-congested path. DCMPTCP [31] and DC^2-MTCP [32] propose some techniques to further optimize MPTCP. Other flow-level schemes include AuTO [29], FlowBender [28], SOFIA [30], VMS [75], Hedera [34], Mahout [35], MicroTE [36] and LocalFlow [27].

3) Flowlet-level schemes. Flowlet-level load balancing schemes [1], [38]–[43] make a trade-off between packet-level schemes and flow-level schemes in consideration of minimizing reordering and maximizing performance at the same time. Flowlets widely exist in data centers where most applications send traffic in on-off patterns [2], [76], [77]. CONGA [41] designs a distributed algorithm to obtain global congestion information in leaf-spine topologies, and assigns each flowlet to the least congested path at leaf switches. LetFlow [1] randomly picks paths for flowlets, and lets their elasticity naturally balance the traffic on different paths. Other flowlet-level schemes include FLARE [38], CLOVE [42], HULA [78], DumbNet [40], and MLAB [39].

Existing flowlet-level schemes use a flowlet table to detect flowlets. Each table entry consists of a `next_hop` and a `timestamp`. In CONGA [41] and LetFlow [1], the `timestamp` is replaced with two bits, and they use a separate process to periodically clean the entries. This table must be very large to keep the collision rate small. Such a huge table incurs heavy memory burden when deployed on hardware platforms where on-chip memory is precious. By contrast, sketch is a compact data structure that uses small memory to perform various measurement tasks [58], [79]–[81]. Typical sketches include CM [82], CU [83], Count [84], CSM [85], and more [86]–[89]. We can use sketches to detect and schedule flowlets in real time, which is still an open area.

III. The BalanceSketch Algorithm

In this section, we propose BalanceSketch to efficiently detect the start of each FlowBurst and select the next hop for the detected FlowBursts. BalanceSketch can be deployed on switches to make load balancing decisions. Next, we first present a strawman solution to detect FlowBursts in § III-A. Then, we introduce the rationale of BalanceSketch in § III-B. We show the data structure and workflow of BalanceSketch in § III-C and § III-D, respectively. We demonstrate how BalanceSketch handles different traffic patterns in § III-E.

A. A Strawman Solution

One strawman solution to find FlowBursts is to first identify all flowlets using existing methods, and then check whether each identified flowlet is a FlowBurst. In the first step, same as existing solutions [38]–[40], we use a `timestamp` array to find flowlets. As shown in Figure 1, the interval between the current time and the last arrival time of f_2 exceeds δ, so we report the packet of f_2 as the start of a flowlet. In the second step, we use a hash table with many buckets to detect FlowBursts. Each bucket maintains a flow ID and the recent speed of the flow. For a flowlet of flow f_i detected in step one, we map f_i into one bucket in the hash table. If another flowlet is already in this bucket and its speed is slow ($< \mathcal{V}$), we replace it with f_i. As shown in Figure 1, for the detected flowlet of f_2, its mapped bucket is taken by f_3 and the speed of f_3 is slow ($< \mathcal{V}$), so we replace f_3 with f_2. This solution is simple and easy to deploy. However, it is memory inefficient because it records the information of all flowlets, whereas most of which are redundant information for unnecessary flowlets. The ideal goal is keeping only FlowBursts while evicting all unnecessary flowlets.

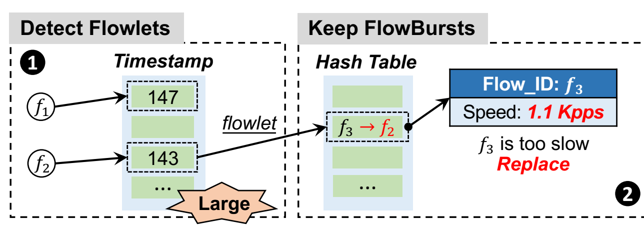


Fig. 1: A strawman solution to detect FlowBursts (δ=5ms, t_{now}=150ms, $\mathcal{V}$=1.5Kpps).

B. Rationale of BalanceSketch

The design of BalanceSketch considers two dimensions of *doing less*: 1) Different from the above strawman solution, we manage to maintain only FlowBursts and evict unnecessary flowlets. 2) We identify most rather than all FlowBursts, in exchange for the simplicity of our data structure and its operations. Besides doing less, we have another design technique: *follower approximation*. Ideally, when the first packet of a FlowBurst arrives, we should know and manipulate it immediately. Obviously, it is almost impossible to immediately assert a flowlet as FlowBurst when it just starts, but it is not hard to assert a FlowBurst when it ends. Instead of manipulating a FlowBurst FB_i, we make a *follower approximation* by manipulating the BurstFollower: the flowlet immediately following FB_i. The rationale is that BurstFollower is a potential FlowBurst, incurring a risk of load imbalance. Interestingly, we find this approximation achieves similar performance to the ideal solution. Consider a typical traffic pattern: FlowBurst, FlowBurst, FlowBurst, FlowBurst, $\cdots$. Ideally, we can manipulate each FlowBurst; Approximately, we manipulate all FlowBurst except the first one. More interesting patterns are provided in § III-E.

C. Data Structure

As shown in Figure 2, the data structure of BalanceSketch is an array of l buckets. Let $\mathcal{B}[i]$ be the i^{th} bucket. Each packet of flow f_i is mapped into one bucket $\mathcal{B}[h(f_i)]$ through a hash function $h(.)$. Each bucket consists of four fields: 1) A `flow_ID` field $\mathcal{B}[i].ID$ records the ID of the flow mapped into this bucket, and we call the flow in the bucket as the *residing flow*. 2) A `vote` field $\mathcal{B}[i].vote$ decides which flow should be stored in this bucket. 3) A `timestamp` field $\mathcal{B}[i].time$ records the arrival time of the last packet of the residing flow. 4) An `next_hop` field $\mathcal{B}[i].nexthop$ records the next hop. For the flow resided in $\mathcal{B}[i]$, if $\mathcal{B}[i].nexthop$ is not $Null$, we forward the flow through this next hop. Otherwise, we forward it using ECMP [25] mechanism: forwarding it through the next hop hashed by its 5-tuple. For the flows not resided in BalanceSketch, we also forward them using ECMP. All fields in the data structure are initialized to 0 or $Null$.

D. Workflow

The pseudo-code of the workflow is shown in Algorithm 1. For an incoming packet p_c of flow f_i at time t_{now}, BalanceSketch takes two phases to process it: *insertion* and *forwarding*. In the *insertion* phase, BalanceSketch inserts f_i into one bucket. In the *forwarding* phase, BalanceSketch selects the appropriate next hop to forward this packet. Next, we explain the two phases in details.

Insertion: First, we compute the hash function $h(f_i)$ to map f_i into the bucket $\mathcal{B}[h(f_i)]$, and try to insert it. There are three cases as follows.

Case 1: If $\mathcal{B}[h(f_i)]$ is empty or $t_{now} - \mathcal{B}[h(f_i)].time > \Delta$, where Δ is the predefined flow timeout threshold to identify

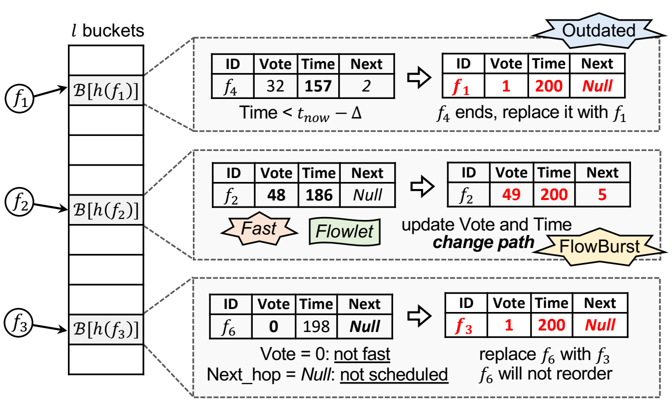


Fig. 2: Examples of BalanceSketch (t_{now}=200ms, Δ=30ms, δ=5ms, $\mathcal{F}$=30).

Algorithm 1: Workflow of BalanceSketch

Input: A packet with timestamp t_i of flow f_i
Output: The next port to send this packet
// Insert the packet into BalanceSketch
if $\mathcal{B}[h(f_i)]$ *is empty* ***or*** $t_i - \mathcal{B}[h(f_i)].time > \Delta$ **then**
  $\mathcal{B}[h(f_i)] \leftarrow \langle f_i, 1, t_i, Null \rangle$;
else if $\mathcal{B}[h(f_i)].ID = f_i$ **then**
  if $\mathcal{B}[h(f_i)].vote > \mathcal{F}$ ***and*** $t_i - \mathcal{B}[h(f_i)].time > \delta$ **then**
    $\mathcal{B}[h(f_i)].nexthop \leftarrow$ the randomly picked next hop;
  $\mathcal{B}[h(f_i)].vote \mathrel{+}= 1$;
  $\mathcal{B}[h(f_i)].time \leftarrow t_i$;
else if $\mathcal{B}[h(f_i)].ID \neq f_i$ **then**
  if $\mathcal{B}[h(f_i)].vote > 0$ **then**
    $\mathcal{B}[h(f_i)].vote \mathrel{-}= 1$;
  if $\mathcal{B}[h(f_i)].vote = 0$ ***and*** $\mathcal{B}[h(f_i)].nexthop = Null$ **then**
    $\mathcal{B}[h(f_i)] \leftarrow \langle f_i, 1, t_i, Null \rangle$;
// Select the next hop to forward the packet
if $\mathcal{B}[h(f_i)].ID = f_i$ ***and*** $\mathcal{B}[h(f_i)].nexthop \neq Null$ **then**
  return $\mathcal{B}[h(f_i)].nexthop$;
else
  return $ECMP\_next\_port(f_i)$;

---

whether a flow ends, we just insert flow f_i into $\mathcal{B}[h(f_i)]$. Specifically, we set $\mathcal{B}[h(f_i)]$ to $\langle f_i, 1, t_{now}, Null \rangle$, where "$Null$" means forwarding flow f_i through ECMP. In this case, $t_{now} - \mathcal{B}[h(f_i)].time > \Delta$ means the previous resided flow is outdated.

Case 2: If $\mathcal{B}[h(f_i)]$ is not empty and f_i is the residing flow, we check whether this packet is the start of a FlowBurst. Specifically, we check whether $t_{now} - \mathcal{B}[h(f_i)].time > \delta$ and $\mathcal{B}[h(f_i)].vote > \mathcal{F}$ are both $true$, where δ is the flowlet threshold and $\mathcal{F}$ is a predefined voting threshold for identifying FlowBursts. If so, it means that the previous flowlet of f_i is a FlowBurst and just ends, and a new flowlet just starts. The new flowlet is potentially a FlowBurst, and thus we manipulate it by *randomly picking* a next hop and update $\mathcal{B}[h(f_i)].nexthop$. Finally, we increment $\mathcal{B}[h(f_i)].vote$ by one and update $\mathcal{B}[h(f_i)].time$ to the current time t_{now}. Note that *randomly picking* a next hop is one design choice, and we can also choose the least loaded next hop or use the "power of two choices" techniques [90].

Case 3: If $\mathcal{B}[h(f_i)]$ is not empty and f_i' is the residing flow where $f_i' \neq f_i$, we decrement $\mathcal{B}[h(f_i)].vote$ by one if $\mathcal{B}[h(f_i)].vote > 0$. Afterwards, if $\mathcal{B}[h(f_i)].vote = 0$ and $\mathcal{B}[h(f_i)].nexthop = Null$, we replace f_i' with f_i. Specifically, we set $\mathcal{B}[h(f_i)]$ to $\langle f_i, 1, t_{now}, Null \rangle$. Note that if $\mathcal{B}[h(f_i)].vote = 0$ but $\mathcal{B}[h(f_i)].nexthop \neq Null$, we do not immediately evict f_i', and will evict it only when it is outdated (the flow timeout threshold Δ) in Case 1. In this way, the

FlowBursts in BalanceSketch will not be frequently replaced, and thus the number of manipulated flow decreases. This is consistent with our design philosophy of doing less.

Forwarding: After inserting f_i into BalanceSketch, we select the next hop to forward the incoming packet p_c. If f_i is the residing flow and $\mathcal{B}[h(f_i)].nexthop \neq Null$, which means that f_i is experiencing a FlowBurst, we forward p_c through $\mathcal{B}[h(f_i)].nexthop$. Otherwise, we forward p_c using ECMP.

Example settings (Figure 2): We use three examples to illustrate the workflow of BalanceSketch, where the three packets of flow $f_1 \sim f_3$ arrive simultaneously at time $t = 200$ms, the *flow timeout threshold* Δ is 30ms, the *flowlet threshold* δ is 5ms, and the *voting frequency threshold* $\mathcal{F}$ is 30.

Example 1 (upper of Figure 2): When a packet of f_1 arrives, it is mapped into bucket $\mathcal{B}[h(f_1)]$. Since $t - \mathcal{B}[h(f_1)].time > \Delta$, we think the residing flow f_4 has ended and replace it with f_1. Since $\mathcal{B}[h(f_1)].nexthop = Null$, we forward the packet using ECMP.

Example 2 (center of Figure 2): When a packet of f_2 arrives, it is mapped into bucket $\mathcal{B}[h(f_2)]$. Since bucket $\mathcal{B}[h(f_2)]$ is not empty and f_2 is the residing flow, we check whether this packet is the start of a FlowBurst. Since $t - \mathcal{B}[h(f_2)].time > \delta$ and $\mathcal{B}[h(f_2)].vote > \mathcal{F}$ are both *true*, we think a previous FlowBurst of f_2 just ends, and the new flowlet has high probability to be a FlowBurst. Thus, we manipulate the new flowlet by changing $\mathcal{B}[h(f_2)].nexthop$ to a randomly picked next hop. We increment $\mathcal{B}[h(f_2)].vote$ by one and update $\mathcal{B}[h(f_2)].time$ to t_{now}. Finally, since f_2 is the residing flow and $\mathcal{B}[h(f_2)].nexthop \neq Null$, we forward the packet through $\mathcal{B}[h(f_2)].nexthop$.

Example 3 (lower of Figure 2): When a packet of f_3 arrives, it is mapped into bucket $\mathcal{B}[h(f_3)]$. Since bucket $\mathcal{B}[h(f_3)]$ is not empty and f_3 is not the residing flow, we decrement $\mathcal{B}[h(f_3)].vote$ by one. Afterwards, since $\mathcal{B}[h(f_3)].vote = 0$ and $\mathcal{B}[h(f_3)].nexthop = Null$, we replace the residing flow f_6 with f_3. Since $\mathcal{B}[h(f_3)].nexthop = Null$, we forward the packet using ECMP.

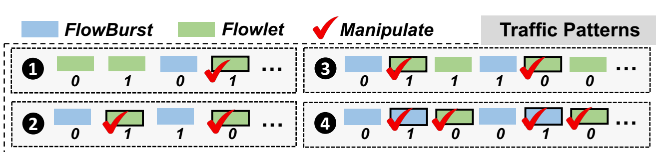


Fig. 3: Examples of typical traffic patterns.

E. Handling Different Traffic Patterns

We take four typical traffic patterns as examples and explain how BalanceSketch handles them, validating our FlowBurst *follower approximation* technique achieves similar load balance performance than the ideal solution of manipulating each FlowBurst at start. For other patterns, they can be regarded as the extension or combination of the four example patterns. For example, the pattern of FlowBurst1, flowlet1, flowlet2, flowlet3, FlowBurst2, flowlet4, flowlet5, flowlet6, $\cdots$ can be regarded as an extension of the third example. In our examples, all FlowBursts/flowlets belong to the same flow. Suppose the default next hop is 0, and the backup next hop is 1.

Pattern 1 (upper-left of Figure 3): This pattern consists of continuous flowlets mixed by a FlowBurst. BalanceSketch manipulates the BurstFollower (the flowlet bounded by black-box in the figure), achieving the same load balance performance as the ideal solution that manipulates each FlowBurst. Note that in this case, manipulating other flowlets (*i.e.*, changing the next hop) benefits little for load balance. BalanceSketch does not manipulate them and manages to achieve *least change* of the next hop. Since there is no frequent manipulation, BalanceSketch minimizes packet reordering. This idea is consistent with our design philosophy of doing less.

Pattern 2 (lower-left of Figure 3): This pattern consists of FlowBurst1, flowlet1, FlowBurst2, flowlet2, $\cdots$. BalanceSketch changes the next hop for each flowlet, and the following FlowBurst is forwarded through the same next hop of the previous flowlet. It achieves similar performance as the ideal solution that manipulates each FlowBurst.

Pattern 3 (upper-right of Figure 3): This pattern consists of FlowBurst1, flowlet1, flowlet2, FlowBurst2, flowlet3, flowlet4, $\cdots$. BalanceSketch changes the next hop for each BurstFollower (*e.g.*, flowlet1), and forwards following flowlet2 and FlowBurst2 through the same next hop. The next hops of BalanceSketch are $\langle 0, 1, 1, 1, 0, 0, \cdots \rangle$, while that of the ideal solution are $\langle 1, 1, 1, 0, 0, 0, \cdots \rangle$. Both BalanceSketch and the ideal solution select one next hop for every two flowlets and one FlowBursts, and thus they have similar performance.

Pattern 4 (lower-right of Figure 3): This pattern consists of FlowBurst1, FlowBurst2, flowlet1, FlowBurst3, FlowBurst4, flowlet2, $\cdots$. BalanceSketch manipulates each latter FlowBurst and each flowlet, and its next hops are $\langle 0, 1, 1, 1, 0, 0, \cdots \rangle$. It achieves similar performance as the ideal solution with the next hops of $\langle 1, 1, 0, 1, 1, 0, \cdots \rangle$.

IV. The BurstBalancer System

A. Overview of BurstBalancer

BurstBalancer deploys BalanceSketch on switches to detect and make forwarding decisions for each FlowBurst. As shown in Figure 4, we deploy one BalanceSketch on each edge switch and let it process all packets arriving from the line side. Given an incoming packet, we first insert its flow ID into one bucket in our BalanceSketch. We check whether the packet is the start of a FlowBurst. If so, we change the next hop of this flow by randomly picking a next hop. Afterwards, we forward the packet through the recorded `next_hop` if its flow ID is recorded, otherwise, we forward it using ECMP. In this way, BurstBalancer divides large and dense flows into FlowBursts, and distributes them to different paths. And for small flows and low-density flows, BurstBalancer just neglects them and forwards them using ECMP. BurstBalancer achieves good load balancing performance while manipulates less flows at the same time.

B. Testbed Implementation

We have fully implemented a BurstBalancer prototype on a testbed with 4 Edgecore Wedge 100BF-32X switches (with Tofino ASIC) [68] and 16 end-hosts in a Leaf-Spine topology. On each switch, we develop BalanceSketch using P4 language [91]. Next, we first describe the challenges we face when implementing BalanceSketch on programmable switches. Then, we describe the workflow of the hardware version of BalanceSketch and analyze the additional resources used by BalanceSketch.

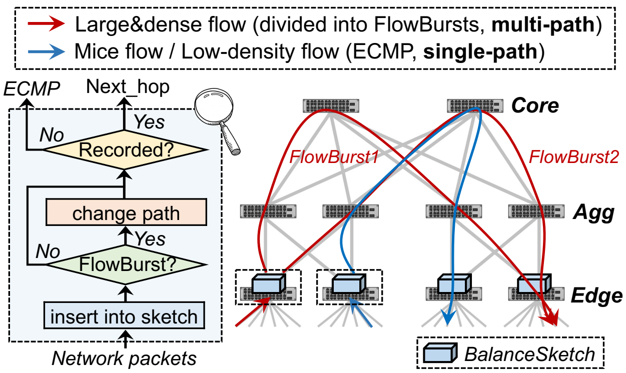


Fig. 4: BurstBalancer overview.

1) Challenges on Programmable Switches:

To process packets at line rate, Tofino switch requires the algorithms running on it to comply with many constraints. Although BalanceSketch is easy to implement on software platforms (*e.g.*, middleboxes, virtual network appliances, *etc.*), when deploying it on hardware, we face the following key challenges.

Resource limitation: We implement BalanceSketch in registers and use the Logical Units in each stage to lookup and update the elements of registers in real time. Recall that each bucket of BalanceSketch consists of four fields (`flow_ID`, `timestamp`, `vote`, and `next_hop`). However, each Stateful ALU can only access one pair of 32-bit elements in each register. Thus, we must divide one bucket into multiple parts and store them in different registers.

Pipeline limitation (I): Tofino switches process packets in a pipelined manner, where each register can only be read or modified once in one pipeline stage. Therefore, each incoming packet can only access each register exactly once, which brings difficulty in clearing the outdated buckets. Due to the first challenge, we have to store the `flow_ID` and `timestamp` of a bucket in two different registers. For each incoming packet, we first check the `flow_ID` register and then update the `timestamp` register if ID matches. However, when ID mismatches and the `timestamp` is outdated (smaller than $t_{now} - \varDelta$), BalanceSketch needs to clear the bucket by setting `flow_ID` to *Null* (Case 1 in § III-D). This backward operation is impossible on Tofino architectures. In our implementation, we consider to use the mirror and recirculate mechanism: once a bucket is identified as outdated, we create a mirror packet and resend it to the ingress port. We use this mirror packet to clear the `flow_ID` register.

Pipeline limitation (II): In the software version of BalanceSketch, if `flow_ID` mismatches and `vote` is decremented to zero, we check whether `next_hop` is *Null*, and evict the *residing flow* f_{old} if so (Case 3 in § III-D). This check operation ensures that the FlowBursts in BalanceSketch are not frequently replaced, and also prevents f_{old} from packet reordering incurred by immediately evicting. However, as explained above, this backward operation cannot be implemented in pipeline. Therefore, in hardware implementation, when `vote` is decremented to zero, we must decide whether to evict the residing flow before checking `next_hop`. To address this issue, we consider dividing BalanceSketch into two parts: a selector and a scheduler. The selector detects FlowBursts and informs the scheduler to schedule them. And the scheduler maintains the next hop information for all scheduled flows. Once a flow is selected to schedule and enters the scheduler, it will be kept until ends. In this way, we approximately implement the software operation of BalanceSketch in a pipelined manner.

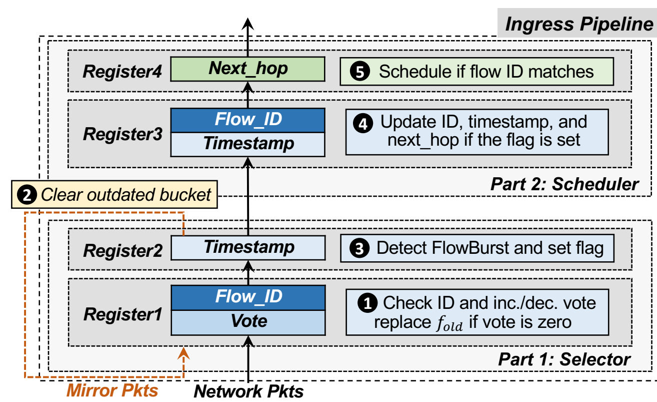


Fig. 5: BalanceSketch on programmable switch.

2) Workflow:

As shown in Figure 5, the workload of BalanceSketch has two parts: a selector and a scheduler. The selector detects FlowBursts and selects the flows to be scheduled. The scheduler keeps the next hop information of the scheduled flows. Both the two parts are implemented in the ingress pipeline.

Selector: Each bucket in selector consists of three fields: `flow_ID`, `vote`, and `timestamp`. The selector uses two registers, where `flow_ID`s and `vote`s are implemented in one register, and `timestamp`s in another. For each incoming packet of f_i, we first check and update the hashed `flow_ID` and `vote` in the first register, *i.e.* increment `vote` if ID matches and decrement it otherwise. If `vote` is decremented to zero, we replace `flow_ID` with f_i. Then we access the hashed `timestamp` in the second register: 1) We check whether the bucket is outdated, *i.e.*, check whether the time gap exceeds $\varDelta$. If so, we create a mirror packet and use it to clear the bucket. 2) We check whether the packet is the start of a FlowBurst, *i.e.*, check whether ID matches, `vote` exceeds $\mathcal{F}$, and time gap exceeds δ. If so, we inform the scheduler to manipulate this flow by setting a temporary variable `sch_flag`. 3) We finally update the `timestamp` to the current time t_{now} if ID matches.

Scheduler: Each bucket in scheduler consists of three fields: `flow_ID`, `timestamp`, and `next_hop`. The scheduler also uses two registers, where `flow_ID` and `timestamp` are implemented in one register, and `next_hop` in another. For each incoming packet of f_i, if it is the start of a FlowBurst, *i.e.*, `sch_flag` is set, we try to update the scheduler: we check the hashed `flow_ID` and `timestamp`. If ID matches or the `timestamp` is outdated (smaller than $t_{now} - \Delta$), we update `flow_ID` to f_i, `timestamp` to t_{now}, and `next_hop` to a randomly chosen next hop. Finally, if the `flow_ID` is f_i, we forward the packet through `next_hop`. Otherwise, we forward the packet using ECMP.

*3) **Hardware Resources Utilization**:*
We show the utilization of different types of hardware resources in Table II. We can see that the average resources usage is less than 10% across all resources, except for Stateful ALUs, which is used for accessing registers and performing transactional read-test-write operations on BalanceSketch. We implement BalanceSketch in 9 stages on Tofino switch: 4 stages for the selector and 2 stages for the scheduler. In addition, we use 3 stages to implement the basic functions of the switch, such as route matching and packet forwarding.

TABLE II: H/W resources used by BalanceSketch.

Resource	Usage	Percentage
Hash Bits	390	7.81%
SRAM	92	9.59%
Map RAM	26	4.51%
TCAM	0	0%
Stateful ALU	13	27.08%
VLIW instr	16	4.17%
Match Xbar	109	7.10%

V. Experimental Results

We extensively evaluate BurstBalancer **(BB)** with testbed experiments (§ V-C) and event-level simulations (§ V-B). We also evaluate the accuracy of BalanceSketch (§ V-A1) and the load balance performance of BurstBalancer on a single switch (§ V-A2). Our experiments aim to answer the following questions.

- **Can BalanceSketch accurately detect FlowBursts?** We implement BalanceSketch using C++ and evaluate its accuracy. The results show that BalanceSketch achieves about 90% Recall Rate in finding FlowBursts. (§ V-A1)
- **Can BurstBalancer manipulate less flows to balance the traffic?** We evaluate the load balance performance of BurstBalancer on a single switch, confirming that compared to LetFlow [1], BurstBalancer manipulates 58× less flows while better balances the traffic. (§ V-A2)
- **In symmetric topologies, can BurstBalancer better balance the traffic?** We extensively evaluate BurstBalancer using two simulation platforms. As a whole, BurstBalancer achieves 5%∼35% better FCT than state-of-the-art LetFlow [1] and DRILL [16] in symmetric topologies. (§ V-B)
- **In asymmetric topologies, can BurstBalancer better balance the traffic?** We evaluate BurstBalancer on a small-scale testbed with asymmetry built with Tofino switches [68]. The results show that BurstBalancer achieves up to 30× better FCT than LetFlow and up to 6.4× better FCT than WCMP [26]. (§ V-C)

Metrics: We use flow completion time (FCT) as the primary performance metric. In certain experiments, we also consider the statistics of the queue lengths across ports and the packet reordering ratio. We use the Recall Rate (RR) to evaluate the accuracy of BalanceSketch in finding FlowBursts.
Workloads: We use two realistic workloads and one synthetic workload in our experiments: 1) Web search workload [92] from a production cluster running web search services, where the average flow size is $\sim 2.5 \times 10^6$ bytes; 2) RPC workload [93] that contains many small flows, where the average flow size is $\sim 2 \times 10^2$ bytes; 3) Synthetic workload that is of heavy-tailed distribution, where the average flow size is ~ 30 packets. The traffic distribution is shown in Figure 6. All the four workloads are heavy-tailed: a small fraction of large flows contribute to most traffic.

A. CPU Experiments

*1) **Accuracy in Finding FlowBursts**:*
We evaluate the accuracy of BalanceSketch in finding FlowBursts under small memory usage.
Platform and implementation: We conduct the experiments on an 18-core CPU server (Intel i9-10980XE) with 128GB DDR4 memory and 24.75MB L3 cache. We use C++ to implement the strawman solution in § III-A and BalanceSketch. In these experiments, our aim is to only detect FlowBursts and do not schedule them, so we do not implement the `next_hop` field in BalanceSketch. BalanceSketch evicts a recorded flow once its `vote` field is decremented to zero.
Dataset: We use the CAIDA [94] anonymized IP traces collected in 2018 from high-speed monitors on a commercial backbone link. The dataset contains about 6M packets belonging to 0.9M different flows. We set the flowlet threshold $\delta = 90\mu s$ and accelerate the traffic speed, so that there are about 17,000 active flowlets on average. We set the speed lower bound of the ground-truth FlowBurst $\mathcal{V}$ to the 70^{th} percentile of the speed of all flowlets. And we set the size threshold of FlowBurst as the size of the 200^{th} flowlets with speed larger than $\mathcal{V}$ ($k = 200$).
Accuracy of basic BalanceSketch (Figure 7): *We find that the RR of BalanceSketch greatly outperforms the strawman solution.* Compared to the strawman solution, RR of BalanceSketch is about 30% higher on average. When using 90KB of memory, BalanceSketch achieves about 90% RR in finding FlowBursts. The results are consistent with our analysis in § III-A. The main reason is that the strawman solution records information of all flowlets, most of which are unnecessary flowlets, incurring enormous redundancy. In contrast, BalanceSketch only keeps FlowBursts and discards

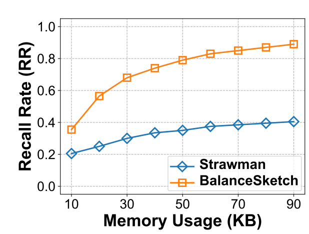
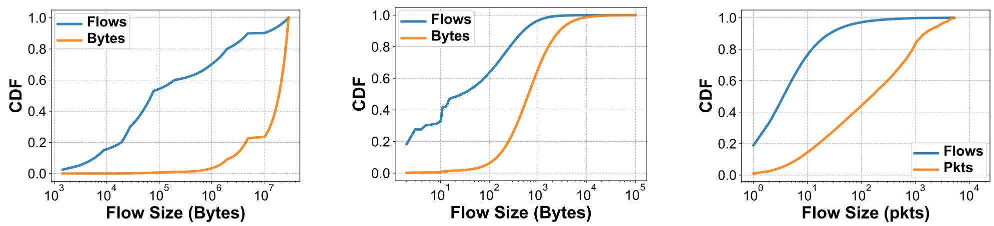


Fig. 7: BalanceSketch accuracy in finding FlowBursts.



(a) Web search. (b) RPC. (c) Synthetic.

Fig. 6: Traffic distributions. The Bytes (Pkts) CDF shows the distribution of traffic bytes (packets) across different flow sizes.

unnecessary flowlets, gaining high memory efficiency. In summary, BalanceSketch well achieves our design goal of accurately identifying FlowBursts using small memory.

*2) **Load Balance Performance on a Single Switch**:* We evaluate the load balance performance of BurstBalancer on single switch and compare it against ECMP [25] and LetFlow [1]. We use C++ to implement the load balancing module of a 128-port switch, on which we deploy the Flowlet Tables (LetFlow) and the BalanceSketchs with different sizes (2K/4K # entries/buckets). The experiments are conducted using the synthetic workload (Figure 6(c)). We measure the standard deviation of the number of packets across all ports, and count the ratio of the manipulated flows.

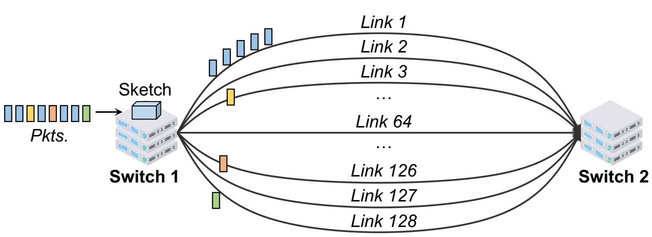
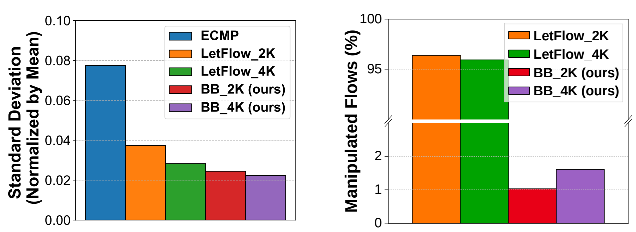


Fig. 8: BurstBalancer on a single switch.

Implementation, traffic, and setting: We use C++ to implement a chip model of a 128-port switch, on which we deploy the Flowlet Tables (LetFlow) and the BalanceSketchs with different sizes (2K/4K # entries/buckets). As shown in Figure 8, in our setting, there are two switches connected by 128 links. We generate the traffic according to the synthetic workload at switch 1. We measure the traffic distribution across the 128 links, and count the reordering packets at switch 2.



(a) Standard deviation of # packets at all ports (normalized by mean). (b) Ratio of manipulated flows.

Fig. 9: Performance of BurstBalancer on single switch: load distribution across all ports, and ratio of manipulated flows.

Load distribution across all ports (Figure 9(a)): *We find that compared to LetFlow, BurstBalancer better balances the traffic using smaller memory.* The results show that the standard deviation of BurstBalancer using 2K buckets is smaller than LetFlow using 4K entries. This is because due to the limited memory and the large number of concurrent flows, LetFlow inevitably regards multiple flows as one, leading the number of detected flowlets decreases a lot. In other words, the large volume of concurrent flows makes LetFlow harder to divide flows into flowlets, resulting in more unbalanced load.

Ratio of manipulated flows (Figure 9(b)): *We find that compared to LetFlow, BurstBalancer manipulates 58 times less flows while achieves better load balance performance.* The results show that the manipulated flows of BurstBalancer is 1.0 %$\sim$1.65%, while that of LetFlow is $> 95\%$. Note the the load balance performance of BurstBalancer_2K is better than LetFlow_4K.

Load distribution for high-density traffic (Figure 10): To better demonstrate the advantages of our BurstBalancer over LetFlow, we use the synthetic workload to create a high-density traffic model, in which a packet arrives in each clock cycle. We repeat the experiments using LetFlow_4K and BurstBalancer_2K. The results show that the performance of LetFlow and ECMP is almost the same, because the high-density traffic makes it difficult for LetFlow to detect flowlets, and thus LetFlow degenerates into ECMP. BurstBalancer can still well balance the traffic since it only manipulates critical flowlets and ignores abundant unnecessary flowlets.

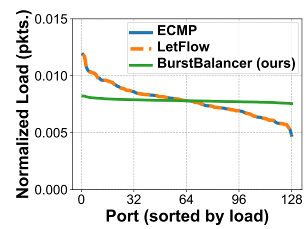
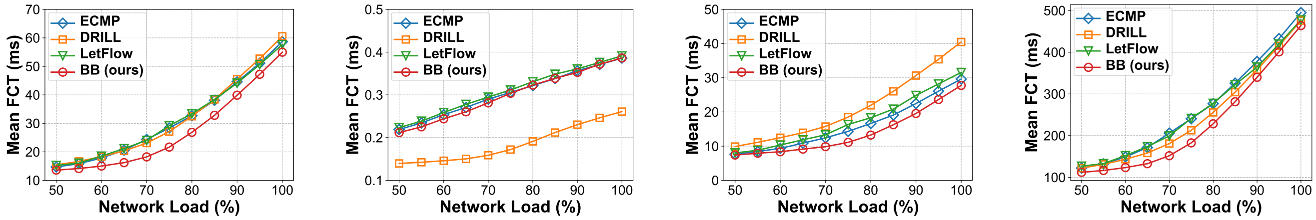


Fig. 10: Load distribution across ports.

B. Event-level Simulations (NS-2)

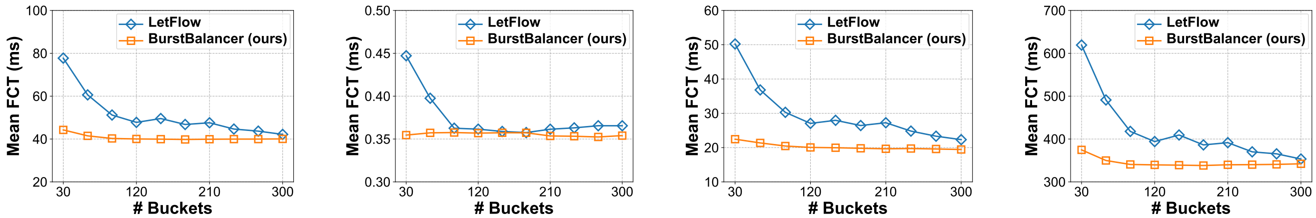
We evaluate BurstBalancer using an event-level network simulator, Network Simulator 2 (NS-2) [69], in large-scale symmetric topologies, where we compare BurstBalancer against ECMP [25], DRILL [16], and LetFlow [1] under different network loads. We also evaluate the performance of BurstBalancer and LetFlow using tables of different sizes, validating the memory efficiency of BurstBalancer.

Topology and traffic: We conduct the experiments in a two-tier Leaf-Spine topology consisting of 8 spine switches and 8 leaf switches. Each leaf switch is connected to 16 servers. All links run at 10Gbps. Here, we have a convergence rate of 2 at



(a) Average FCT of all flows. (b) FCT of small flows (<100KB). (c) FCT of medium flows. (d) FCT of large flows (>10MB).

Fig. 11: NS-2 simulation results: FCT statistics under different network loads.



(a) Average FCT of all flows. (b) FCT of small flows (<100KB). (c) FCT of medium flows. (d) FCT of large flows (>10MB).

Fig. 12: NS-2 simulation results: FCT statistics of LetFlow and BurstBalancer using tables of different sizes.

the leaf level, which is common in modern data centers [1], [2]. We configure 90% of the network bandwidth to deliver the web search workload (Figure 6(a)), and the rest to deliver the RPC workload (Figure 6(b)) as background traffic.

Setting: For BurstBalancer and LetFlow, we configure the BalanceSketch/Flowlet Table to have 250 buckets/entries by default. We set the flowlet threshold $\delta = 200\mu s$, set the flow timeout threshold $\varDelta = 50ms$, and set the voting threshold $\mathcal{F} = 0$.

FCT *v.s.* network load (Figure 11): *We find that the overall average FCT of BurstBalancer is always lower than ECMP, DRILL, and LetFlow under different network loads.* As shown in Figure 11(a), as network loads vary, the overall average FCT of BurstBalancer changes from 13.6*ms* to 54.9*ms*, while that of ECMP, DRILL, and LetFlow changes from 14.7*ms*, 15.4*ms*, and 15.3*ms* to 58.6*ms*, 60.6*ms*, and 57.7*ms*, respectively. In summary, BurstBalancer achieves up to ∼25.2%, ∼20.1%, and ∼25.8% lower overall average FCT than ECMP, DRILL, and LetFlow, respectively. We further study the average FCT of small flows ($<$ 100KB), medium flows (0.1∼10MB), and large flows ($>$ 10MB) in Figure 11(b)-11(d). The results show that for small flows, DRILL has the lowest average FCT because it balances the traffic at the finest granularity. But for medium flows and large flows, the average FCT of DRILL is high because it suffers significant packet reordering. BurstBalancer always achieves the lowest average FCT for medium flows and large flows among all schemes.

FCT *v.s.* number of buckets/entries (Figure 12): *We find that the overall average FCT of BalanceSketch always outperforms LetFlow under different table sizes.* The experiments are conducted under 90% network loads. As shown in Figure 12(a), as the number of buckets varies, the overall average FCT of BurstBalancer changes from 44.2*ms* to 40.1*ms*, while that of LetFlow changes from 77.7*ms* to 42.2*ms*. The results show that the gap between BurstBalancer and LetFlow becomes larger as the number of buckets decreases. This is because LetFlow cannot accurately divide flows into flowlets under small memory usage. In summary, BurstBalancer achieves up to ∼43.1% lower overall average FCT than LetFlow. We further study the average FCT of flows of different sizes in Figure 12(b)-12(d). The results are similar to that of the overall average FCT.

C. Testbed Experiments

As described in § IV-B, we build a small-scale testbed in an asymmetric topology, on which we compare BurstBalancer against WCMP [26], and LetFlow [1].

Topology and traffic: As shown in Figure 13, we use a two-tier Leaf-Spine topology consisting of 2 spines switches and 2 leaf switches, each of which is connected to 8 servers. All links run at 40Gbps. We fail one of the two links between a leaf and a spine to create asymmetry. We use a client-server program to generate dynamic traffic [95], where the client application generates requests through persistent TCP connections based on a Poisson process, and the server application responds with the requested data. On each leaf, we configure 6 servers to generate requests to 6 servers under another leaf according to the web search workload (Figure 6(a)). We configure the other 2 servers to generate single-packet requests to 2 servers under another leaf. The single-packet requests are used as background traffic to improve the number of concurrent flows in the network. We configure the bandwidth usage of the single-packet traffic as ∼ 5Gbps.

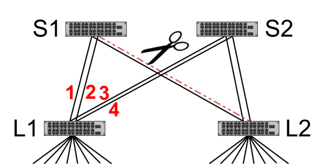
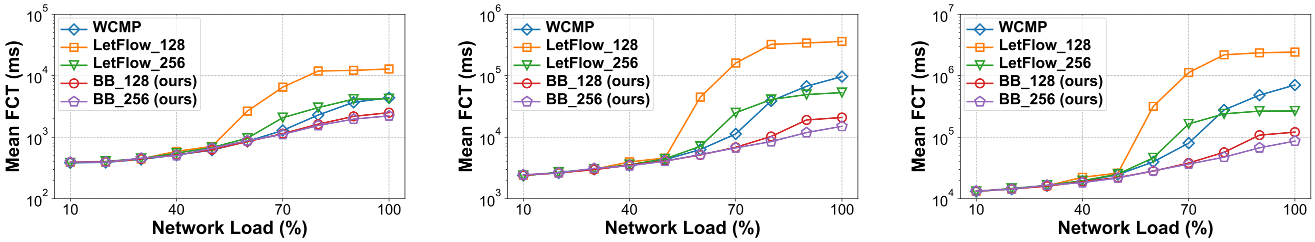


Fig. 13: Testbed topology with asymmetry. All links run at 40 Gbps.

Setting: For BurstBalancer and LetFlow, we configure BalanceSketch/Flowlet Table to have 128 or 256 buckets/entries.



(a) FCT of small flows (<100KB). (b) FCT of medium flows (0.1∼10MB). (c) FCT of large flows (>10MB).

Fig. 14: Testbed results: FCT statistics under different network loads in asymmetric topology.

For WCMP, we configure the weighted cost only according to the localized link status of the switch. We set the flowlet threshold $\delta = 500\mu s$, set the flow timeout threshold $\varDelta = 50ms$, and set the voting threshold $\mathcal{F} = 0$.

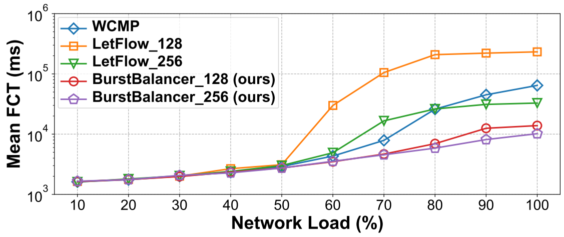
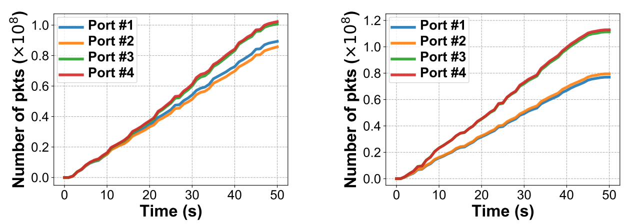


Fig. 15: Testbed results: Average FCT of all flows under different loads in asymmetric topology.

FCT *v.s.* network load (Figure 14-15): *We find that in asymmetric typologies, the overall average FCT of BurstBalancer is always better than WCMP and LetFlow under different network loads.* As shown in Figure 15, as network loads vary, the overall average FCT of WCMP changes from 1.62*ms* to 64.4*ms*. The overall average FCT of BurstBalancer using BalanceSketch of 128 buckets and 256 buckets change from 1.63*ms* and 1.65*ms* to 13.8*ms* and 10.2*ms*, respectively. And the overall average FCT of LetFlow using Flowlet Table of 128 entries and 256 entries change from 1.64*ms* and 1.61*ms* to 232*ms* and 32.8*ms*, respectively. Due to asymmetry, the average FCT has a sudden increase between 50%∼60% network loads. As a whole, the average FCT of BurstBalancer is significantly lower than WCMP and LetFlow, and the BurstBalancer using 128 buckets and 256 buckets have similar performance. LetFlow has higher FCT than BurstBalancer because when using Flowlet Table of 128/256 entries, each table entry will be collided by many flows, making it harder to detect flowlets. Note that when using 128 table entries, the average FCT of LetFlow is significantly higher than the others. This is because such small memory makes it difficult for LetFlow to detect flowlets, and thus the `next_hops` in the Flowlet Table almost remains unchanged. In LetFlow, each flow is forwarded through the `next_hop` recorded in one of the 128 entries. Since the distribution of the 128 `next_hops` is highly uneven, the load balance performance is bad. We further study the average FCT of flows of different sizes in Figure 14(a)-14(c). The results are similar to that of the overall average FCT.

Forwarding statistics of the four ports in a leaf switch (Figure 16): *We find that in asymmetric topologies, BurstBalancer achieves the traffic distribution closer to the optimal ratio.* We measure the number of forwarded packets of the four fabric ports in a leaf switch (shown in Figure 13) under 90% network loads. In this asymmetric topology, the optimal traffic distribution ratio among `Port#1`∼`Port#4` is 1:1:2:2. As shown in Figure 16(a), for ECMP, the traffic distribution ratio is 1:0.96:1.12:1.14. This ratio is not 1:1:1:1 thanks to the implicit feedback mechanism of persistent connections: the probability of reusing congested connections is small. As shown in Figure 16(b), for BurstBalancer, the traffic distribution ratio is 1:1.03:1.45:1.47. BurstBalancer achieves better traffic distribution because the flows on more congested path are more likely to experience a flowlet timeout, so they have more chance to shift to a less congested path.



(a) Number of pkts (ECMP). (b) Number of pkts (BurstBalancer).

Fig. 16: Testbed results: Number of forwarded packets of four ports in asymmetric topology.

VI. Conclusion

This paper presents BurstBalancer, an efficient load balancing system for data center networks. Based on flowlet, the design philosophy of BurstBalancer is to only manipulate a small amount of critical flowlets. We formally define these critical flowlets as FlowBursts. BurstBalancer proposes a compact sketch algorithm, namely BalanceSketch, to accurately identify and manipulate most FlowBursts under small memory usage. Experiments on a testbed and simulations show that BurstBalancer outperforms state-of-the-art LetFlow in both symmetric and asymmetric topologies, while manipulates less flows at the same time. In the future work, we plan to design a mechanism to automatically adjust the system parameters according to the current traffic characteristics; and we plan to integrate BurstBalancer into real network measurement and management systems.

REFERENCES

[1] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 407–420.

[2] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, "Load balancing in data center networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2324–2352, 2018.

[3] D. Wu, W. Wang, A. Chen, and T. S. E. Ng, "Say no to rack boundaries: Towards a reconfigurable pod-centric dcn architecture," in *Proceedings of the 2019 ACM Symposium on SDN Research*, ser. SOSR '19, 2019, p. 112–118.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, p. 123–137.

[5] B. Yeganeh, R. Durairajan, R. Rejaie, and W. Willinger, "How cloud traffic goes hiding: A study of amazon's peering fabric," in *Proceedings of the Internet Measurement Conference*, ser. IMC '19, 2019, p. 202–216.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[7] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009, pp. 51–62.

[8] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008, pp. 75–86.

[9] P. Gratz, B. Grot, and S. W. Keckler, "Regional congestion awareness for load balance in networks-on-chip," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 203–214.

[10] O. Lysne, T. Skeie, S.-A. Reinemo, and I. Theiss, "Layered routing in irregular networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 1, pp. 51–65, 2005.

[11] D. R. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *Proceedings of the 16th annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2004, pp. 36–43.

[12] R. Hsiao and S.-D. Wang, "Jelly: a dynamic hierarchical p2p overlay network with load balance and locality," in *24th International Conference on Distributed Computing Systems Workshops, 2004. Proceedings.*, 2004, pp. 534–540.

[13] J. Alvarez-Horcajo, D. Lopez-Pajares, J. M. Arco, J. A. Carral, and I. Martinez-Yelmo, "Tcp-path: Improving load balance by network exploration," in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, 2017, pp. 1–6.

[14] M. J. Freedman and R. Morris, "Tarzan: A peer-to-peer anonymizing network layer," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 193–206.

[15] A. K. Y. Cheung and H.-A. Jacobsen, "Dynamic load balancing in distributed content-based publish/subscribe," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2006, pp. 141–161.

[16] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, p. 225–238.

[17] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, p. 29–42.

[18] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda, "Multi-path transport for {RDMA} in datacenters," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 357–371.

[19] J. Huang, W. Lv, W. Li, J. Wang, and T. He, "Qdaps: Queueing delay aware packet spraying for load balancing in data center," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018, pp. 66–76.

[20] S. Zou, J. Huang, J. Wang, and T. He, "Improving tcp robustness over asymmetry with reordering marking and coding in data centers," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 57–67.

[21] S. Liu, J. Huang, W. Jiang, J. Wang, and T. He, "Reducing flow completion time with replaceable redundant packets in data center networks," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 46–56.

[22] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 49–60.

[23] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "Detail: reducing the flow completion time tail in datacenter networks," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 139–150.

[24] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized" zero-queue" datacenter network," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 307–318.

[25] C. Hopps *et al.*, "Analysis of an equal-cost multi-path algorithm," RFC 2992, November, Tech. Rep., 2000.

[26] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "Wcmp: Weighted cost multipathing for improved fairness in data centers," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[27] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, optimal flow routing in datacenters via local link balancing," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 151–162.

[28] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 149–160.

[29] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, 2018, p. 191–205.

[30] F. De Pellegrini, L. Maggi, A. Massaro, D. Saucez, J. Leguay, and E. Altman, "Blind, adaptive and robust flow segmentation in datacenters," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 10–18.

[31] E. Dong, X. Fu, M. Xu, and Y. Yang, "Dcmptcp: Host-based load balancing for datacenters," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 622–633.

[32] J. Sun, Y. Zhang, X. Wang, S. Xiao, Z. Xu, H. Wu, X. Chen, and Y. Han, "$dc^2 - mtcp$: Light-weight coding for efficient multi-path transmission in data center network," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 419–428.

[33] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 266–277, 2011.

[34] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat *et al.*, "Hedera: dynamic flow scheduling for data center networks." in *Nsdi*, vol. 10, no. 8. San Jose, USA, 2010, pp. 89–92.

[35] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *2011 Proceedings IEEE INFOCOM*, 2011, pp. 1629–1637.

[36] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the seventh conference on emerging networking experiments and technologies*, 2011, pp. 1–12.

[37] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 202–215.

[38] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, p. 51–62, 2007.

[39] F. Fan, B. Hu, and K. L. Yeung, "Routing in black box: Modularized load balancing for multipath data center networks," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1639–1647.
[40] Y. Li, D. Wei, X. Chen, Z. Song, R. Wu, Y. Li, X. Jin, and W. Xu, "Dumbnet: A smart data center network fabric with dumb switches," in *Proceedings of the 13th EuroSys Conference*, ser. EuroSys '18, 2018.
[41] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 503–514.
[42] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford, "Clove: How i learned to stop worrying about the core and love the edge," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016, pp. 155–161.
[43] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 465–478, 2015.
[44] S. Sinha, S. Kandula, and D. Katabi, "Harnessing tcp's burstiness with flowlet switching," in *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*. Citeseer, 2004.
[45] S. Prabhavat, H. Nishiyama, N. Ansari, and N. Kato, "Effective delay-controlled load distribution over multipath networks," *IEEE transactions on Parallel and distributed systems*, vol. 22, no. 10, pp. 1730–1741, 2011.
[46] B. Radunović, C. Gkantsidis, D. Gunawardena, and P. Key, "Horizon: Balancing tcp over multiple paths in wireless mesh network," in *Proceedings of the 14th ACM international conference on Mobile computing and networking*, 2008, pp. 247–258.
[47] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016, p. 129–143.
[48] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with netpoirot," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016, p. 440–453.
[49] J. Gao, N. Yaseen, R. MacDavid, F. V. Frujeri, V. Liu, R. Bianchini, R. Aditya, X. Wang, H. Lee, D. Maltz, M. Yu, and B. Arzani, "Scouts: Improving the diagnosis process through domain-customized incident routing," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, 2020, p. 253–269.
[50] Y. Wu, A. Chen, and L. T. X. Phan, "Zeno: Diagnosing performance problems with temporal provenance," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 395–420.
[51] W. Wang, P. Tammana, A. Chen, and T. S. E. Ng, "Grasp the root causes in the data plane: Diagnosing latency problems with spidermon," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '20, 2020, p. 55–61.
[52] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 419–435.
[53] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17, 2017, p. 78–85.
[54] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: Steering oceans of content to the world," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, p. 418–431.
[55] J. K. Sundararajan, D. Shah, M. Medard, M. Mitzenmacher, and J. Barros, "Network coding meets tcp," in *IEEE INFOCOM 2009*, 2009, pp. 280–288.
[56] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger, "Tcp revisited: A fresh look at tcp in the wild," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '09, 2009, p. 76–89.
[57] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19, 2019, pp. 44–58.
[58] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "Pint: Probabilistic in-band network telemetry," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, 2020, p. 662–680.
[59] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016, p. 58–72.
[60] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao, "Efficient querying and maintenance of network provenance at internet-scale," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010, p. 615–626.
[61] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with netqre," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, 2017, p. 99–112.
[62] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, p. 183–197.
[63] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 202–208.
[64] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 909–921.
[65] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the 2013 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '13)*, 2013, p. 435–446.
[66] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao *et al.*, "Packet-level telemetry in large datacenter networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*, 2015, pp. 479–491.
[67] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Deconstructing datacenter packet transport," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets '12)*, 2012, pp. 133–138.
[68] "Barefoot tofino: World's fastest p4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino/.
[69] "The Network Simulator - ns-2," https://www.isi.edu/nsnam/ns/.
[70] "The source codes related to burstbalancer," https://github.com/BurstBalancer/Burst-Balancer.
[71] A. Kabbani and M. Sharif, "Flier: Flow-level congestion-aware routing for direct-connect data centers," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.
[72] P. Wang, G. Trimponias, H. Xu, and Y. Geng, "Luopan: Sampling-based load balancing in data center networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 133–145, 2019.
[73] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 253–266.
[74] J. Hu, J. Huang, W. Lv, Y. Zhou, J. Wang, and T. He, "Caps: Coding-based adaptive packet spraying to reduce flow completion time in data center," *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2338–2353, 2019.
[75] Z. Li, J. Bi, Y. Zhang, A. B. Dogar, and C. Qin, "Vms: Traffic balancing based on virtual switches in datacenter networks," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, 2017, pp. 1–10.
[76] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," in *Proceedings of the conference on Communications architecture & protocols*, 1991, pp. 133–147.

[77] Z.-L. Zhang, V. J. Ribeiro, S. Moon, and C. Diot, "Small-time scaling behaviors of internet backbone traffic: an empirical study," in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, vol. 3. IEEE, 2003, pp. 1826–1836.
[78] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, 2016, pp. 1–12.
[79] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003, pp. 234–247.
[80] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina, "Detection and identification of network anomalies using sketch subspaces," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006, pp. 147–152.
[81] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005, pp. 398–412.
[82] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[83] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *ACM SIGMCOMM CCR*, vol. 32, no. 4, 2002.
[84] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.
[85] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 5, pp. 1622–1634, 2012.
[86] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.
[87] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14, 2014, pp. 109–118.
[88] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 121–132, 2008.
[89] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, 2018, p. 561–575.
[90] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.
[91] "P4-16 language specification," https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-checksums.
[92] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.
[93] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 221–235.
[94] "The caida anonymized 2016 internet traces." http://www.caida.org/data/overview/.
[95] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling {ECN} in multi-service multi-queue data centers," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 537–549.