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

Zirui Liu, Yikai Zhao, Zhuochen Fan, Tong Yang, *Member, IEEE*, Xiaodong Li, Ruwen Zhang, Kaicheng Yang, Zihan Jiang, Zheng Zhong, Yi Huang, Cong Liu, Jing Hu, Gaogang Xie, *Senior Member, IEEE*, and Bin Cui, *Senior Member, IEEE*

***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 article 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 our balance operations. We have fully implemented BurstBalancer in a small-scale testbed built with Tofino switches, and conducted both large-scale event-level (NS-2) and ESL (electronic system level) simulations. Our results show that BurstBalancer achieves 5%∼35% smaller FCT than LetFlow in symmetric topology and up to 30× smaller FCT in asymmetric topology, while 58× fewer flows suffer from path changing. All related codes are open-sourced at GitHub.**

***Index Terms*—Data center networks, L3 load balancing, sketch, flowlet.**

I. Introduction

A. Background and Motivation

As ENTERPRISES continue to shift services like big-data analytics, web services, and cloud storage into cloud environments, the number of data centers has grown exponentially [2], [3], [4], [5], [6], [7]. Typical data center networks (DCNs) feature symmetric topologies such as Fat-Tree [8] and VL2 [9], which offer multiple equivalent paths between any pair of servers. 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 [10], [11], [12], [13].

Existing L3 load balancing schemes can be broadly classified into three categories. First, packet-level load balancing schemes [14], [15], [16], [17], [18], [19] select a path for each packet to achieve optimal traffic split. However, these schemes are susceptible to packet reordering issues when there is a significant discrepancy in the delay of candidate paths. Second, flow-level load balancing schemes [20], [21], [22], [23], [24], [25], [26] assign a fixed path to all packets within a flow. These schemes avoid packet reordering, but cannot well balance the traffic due to the skewed distribution of flow sizes and hash collisions among large flows [27]. Third, flowlet-level load balancing schemes [2], [28], [29], [30], [31], [32], [33] attempt a compromise between minimizing packet reordering and evenly balancing traffic. In these schemes, the packets of a flow are divided into many groups, with the time interval between two consecutive groups being larger than a predefined threshold δ. Each group of packets is called a flowlet [28]. Flowlet-level schemes select a path for each flowlet, so as to achieve a good trade-off between packet reordering and load balance [3], [33], [34], [35]. However, these schemes cannot precisely detect flowlets using small memory, and make a lot of unnecessary manipulation: 1) The forwarding paths of the flows in these schemes are unfixed and unpredictable, which hampers network measurement and management. 2) Due to the limited memory on hardware and the large number of concurrent flows, they cannot precisely identify all flowlets. 3) They unnecessarily divide small flows into flowlets and distribute them across multiple paths, which contributes little to load balance and rather increases the risk of packet reordering.

While existing load balancing schemes have made excellent contributions, they overlook the *flow-regulation* of the network. “Flow-Regulation” refers to that given any flow, its forwarding path can be easily calculated, and does not change with time. In many existing schemes, the forwarding paths of flows are unfixed and unpredictable. 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 [36], [37], [38], [39], congestion control [40], [41],

Manuscript received 6 November 2022; revised 6 June 2023; accepted 5 July 2023. Date of publication 14 July 2023; date of current version 12 April 2024. This work was supported in part by the National Key R&D Program of China under Grant 2022YFB2901504, and in part by the National Natural Science Foundation of China (NSFC) under Grant U20A20179. Recommended for acceptance by Y. Yang. The preliminary version of this paper titled “BurstBalancer: Do Less, Better Balance for Large-scale Data Center Traffic” is published in the 30th IEEE International Conference on Network Protocols (IEEE ICNP 2022) [1], Lexington, Kentucky, USA, October 30-November 2, 2022. *(Zirui Liu, Yikai Zhao, and Zhuochen Fan are the co-primary authors.) (Corresponding author: Tong Yang.)*

Zirui Liu, Yikai Zhao, Zhuochen Fan, Ruwen Zhang, Kaicheng Yang, Zihan Jiang, Zheng Zhong, and Bin Cui are with the National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China.

Tong Yang and Xiaodong Li are with Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: yangtongemail@gmail.com).

Yi Huang, Cong Liu, and Jing Hu are with Huawei Technologies Company Ltd., Shenzhen 518063, China.

Gaogang Xie is with the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100045, China.

Digital Object Identifier 10.1109/TPDS.2023.3295454

[42], [43], network measurement and management [44], [45], [46], [47], [48], [49], [50], [51], it is often assumed or expected that the forwarding paths of most flows can be obtained easily. For example, the well known 007 system [37] 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 diagnostic results. For another example, the pioneering work using INT for congestion control, HPCC [42], 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. Hence, our goal is to develop a solution that not only effectively balances traffic, but also ensures adherence to the flow regulation principle 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 flowlets that are critical to load balance, namely FlowBursts. In BurstBalancer, most flows follow ECMP [20] and we can easily get their forwarding paths. BurstBalancer devises a sketch, namely BalanceSketch, and deploys it on each switch to detect and make forwarding decisions for FlowBursts. BurstBalancer only needs small on-chip memory to keep critical flowlets (FlowBursts), achieving high memory efficiency and perfectly embracing the skewed flow distribution [52], [53]. Further, BurstBalancer only manipulates the critical flowlets which are very limited in number, minimizing packet reordering and keeping the paths of most flows fixed. 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 *FlowBurst*s.[1] Therefore, we *manipulate only critical flowlets (FlowBursts).* For example, in the IMC data center trace [54] used in our experiments, there are about 27,000 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. In this way, we classify all flowlets into two categories: FlowBursts

[1]A formal definition of FlowBurst is provided in Section II-A.

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 by our BalanceSketch to identify flowlets with 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 a flow into a bucket
d	Number of cells in each bucket in the multi-cell version of BalanceSketch

and unnecessary flowlets,[2] and we only manipulate FlowBursts. Identifying unnecessary flowlets causes a huge memory overhead,[3] and manipulating them only exacerbates network chaos and the risk of packet reordering.

The second dimension of doing less is based on *Observation II:* It is expensive and unnecessary to accurately detect and manipulate all FlowBursts. Hence, we choose to *manipulate most rather than all FlowBursts* for the following reasons. 1) Identifying all FlowBursts is expensive for hardware resources. 2) By manipulating most FlowBursts while leaving other FlowBursts to follow ECMP, we can still attain effective load balancing. 3) Detecting all FlowBursts needs complicated design of data structure. A strawman solution to identify FlowBursts is to first identify 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. We propose a simple data structure, namely BalanceSketch, to track the most relevant, rather than all FlowBursts (See details in Section III) and evict unnecessary flowlets. To the best of our knowledge, we are the first work that applies sketches to the field of L3 load balancing.

We extensively evaluate BurstBalancer on a small-scale testbed and two large-scale simulation platforms. Our testbed consists of 4 Tofino switches [55] and 8 end-hosts in a leaf-spine topology. For simulations, we use both an event-level simulator (NS-2 [56]) and an ESL (electronic system level) simulator (HDCN, which is developed and used by Huawei for years). Our results show that compared to LetFlow [2], BurstBalancer better balances the traffic using smaller memory, while manipulates 58 times fewer flows at the same time. In symmetric topologies, BurstBalancer achieves 5%∼35% smaller FCT (flow completion time) than state-of-the-art LetFlow [2] and DRILL [14]. In asymmetric topologies, BurstBalancer achieves up to 30× smaller FCT than LetFlow and up to 6.4× smaller FCT than WCMP [21]. We also conduct CPU experiments, and results show that BurstBalancer achieves > 90% recall rate in finding FlowBursts with small memory. In addition, we mathematically

[2]Unnecessary flowlets are defined as: 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).

[3]In our experiments, LetFlow [2], a load balancing scheme identifying and manipulating all flowlets, consumes about 10 × more on-chip memory to achieve a similar load balancing performance to our BurstBalancer (see Fig. 15).

derive the ability of BalanceSketch to identify FlowBursts (see Section III-G). All related codes are open-sourced [57].

II. Background and Related Work

In this section, we begin with the problem statement of FlowBurst in Section II-A. Then we discuss the related work of load balance solutions for data center networks in Section 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 $S = \{p_1, p_2, \ldots\}$, 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 $\{p'_1, \ldots, p'_n\}$ 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 $\{p'_1, \ldots, p'_m\}$ 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 adjacent flowlets is large enough ($> \delta$).*

FlowBurst: For a flowlet $\{p'_1, \ldots, p'_m\}$, we define its size as m, and define its speed as $\frac{m}{\Delta T}$, where $\Delta T = t_m - t_1$. This flowlet is a FlowBurst if $\frac{m}{\Delta T} > \mathcal{V}$ 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 classified into three categories: packet-level schemes, flow-level schemes, and flowlet-level schemes. For other hybrid schemes, kindly refer to references [33], [58], [59], [60].

1) Packet-level schemes [14], [15], [16], [17], [18], [19] choose a desirable path for each packet. They achieve ideal splitting ratio at the cost of packet reordering. DRILL [14] makes per-packet decisions at each switch based on local-queue occupancies and randomized algorithms. NDP [15] presents a multipath-aware transport-layer protocol that manipulates each packet, and introduces a handshake mechanism to alleviate reordering. MP-RDMA [16] 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 [19], DeTail [18], and DRB [17].

2) Flow-level schemes [20], [21], [22], [23], [24], [25], [26], [61] 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 [20] uses flow-level hashing to select a path for each flow. ECMP achieves excellent performance when there are only small flows but no large flows [25], [26]. WCMP [21] assigns each path a weighted cost, and distributes the traffic based on the cost. MPTCP [61] splits each TCP flow into several subflows, and assigns each subflow to a non-congested path. AuTO [23] forwards small flows using ECMP, and dynamically changes path, priority, and sending speed for large flows. Other flow-level schemes include FlowBender [22], SOFIA [24], VMS [62], Hedera [25], Mahout [26], MicroTE [63].

3) Flowlet-level schemes [2], [28], [29], [30], [31], [32], [33], [64] 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 [3], [65], [66]. CONGA [31] 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 [2] randomly picks paths for flowlets, and lets their elasticity naturally balance the traffic on different paths. The excellent work Contra [67] builds a system for performance-aware routing based on flowlet switching, which can operate seamlessly over any network topology and routing policies. Other flowlet-level schemes include DASH [64], FLARE [28], HULA [68], and more [29], [30], [32]. A flowlet scheme needs to strike a balance between load balance and packet reordering. A flowlet switching scheme has no danger of packet reordering only when the timeout threshold δ is larger than the maximum latency of the set of parallel paths. In order to avoid packet reordering, the timeout threshold must be set to a large value. However, large timeout threshold will degrade the system to a flow-level scheme. Therefore, the timeout threshold δ should be carefully chosen to achieve good performance.

Existing flowlet-level schemes use a flowlet table to detect flowlets. Each table entry consists of a `next_hop` and a `timestamp`. In CONGA [31] and LetFlow [2], 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 [43], [69], [70]. Typical sketches include CM [71], CU [72], Count [73], and more [74], [75], [76]. 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 first present a strawman solution to detect FlowBursts in Section III-A, and introduce the rationale of BalanceSketch in Section III-B. We show the data structure and workflow of BalanceSketch in Sections III-C and III-D. We demonstrate how BalanceSketch handles different traffic patterns in Section III-E. We propose some optimizations of BalanceSketch in Section III-F. We mathematically analyze the ability of BalanceSketch to identify FlowBursts in Section III-G.

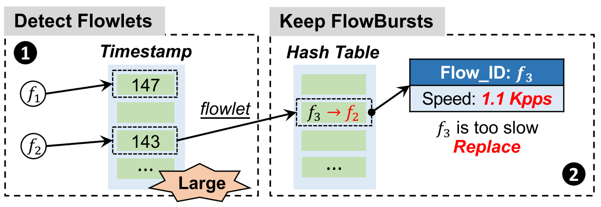


Fig. 1. Strawman solution to detect FlowBursts ($\delta = 5$ ms, $t_{now} = 150$ ms, $\mathcal{V} = 1.5$ Kpps).

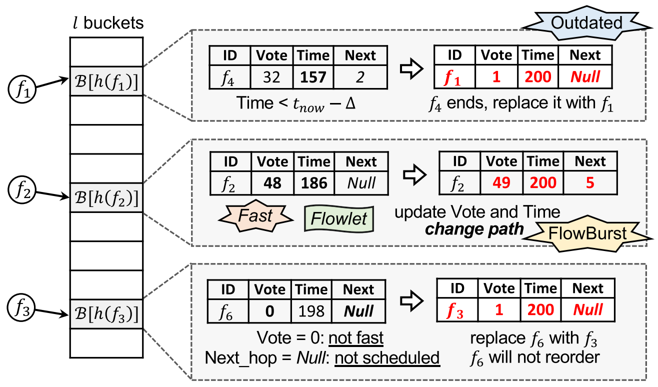


Fig. 2. Examples of BalanceSketch ($t_{now} = 200$ ms, $\Delta = 30$ ms, $\delta = 5$ ms, $\mathcal{F} = 30$).

Finally, we present how to extend BalanceSketch to network measurement in Section III-H.

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. 1) In the first step, same as existing solutions [28], [29], [30], we use a `timestamp` array to find flowlets. For each incoming packet of flow f_i at time t_{now}, we first compute a hash function $h_1(f_i)$ to map the packet to one timestamp. If the gap between t_{now} and `timestamp` is larger than the predefined flowlet threshold δ, we consider the packet as the start of a flowlet. Otherwise, we consider the packet as part of an existing flowlet. At the end of this step, we update the mapped timestamp to t_{now}. As shown in Fig. 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. 2) In the second step, we use a hash table with many buckets to detect FlowBursts, i.e., the flowlets with high speed. 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. Specifically, each bucket in the second step consists of a `Flow_ID` field, a `Start_time` field recording the start time of the flowlet, and a `Counter` field recording the flowlet size. Given an incoming packet of f_i at t_{now}, we first compute hash function $h_2(f_i)$ to map the packet into one bucket. Then, we check whether f_i is recorded in this bucket. If so, we increment the `Counter` by one. Otherwise, if the packet is the start of a flowlet (detected in the first step), we check whether its speed `Counter` / ($t_{now}-$`Start_time`) is below the speed threshold $\mathcal{V}$: If so, we replace the old flowlet with f_i: we set `Flow_ID` to f_i, set `Start_time` to current time t_{now}, and set `Counter` to one. As shown in Fig. 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, including the exact flow IDs and their recent speed, whereas most flowlets are unnecessary flowlets. The ideal goal is keeping only FlowBursts while evicting all unnecessary flowlets.

B. Rationale of BalanceSketch

The design of BalanceSketch embraces two dimensions of *doing less:* 1) Different from the aforementioned 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. Beyond the principle of doing less, we have another design tactic: *follower approximation*. Ideally, upon the arrival of the first packet of a FlowBurst, we should immediately recognize and manipulate it. However, it is almost impracticable to identify a flowlet as a FlowBurst at its onset, but it is not hard to assert a FlowBurst when it ends. Instead of manipulating a FlowBurst, we make a *follower approximation* by manipulating the BurstFollower: the flowlet that immediately succeeds a FlowBurst. The rationale is that BurstFollower is very likely to be 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, ... Ideally, we manipulate each FlowBurst at it commences; Approximately, we manipulate each BurstFollowers, which essentially includes all FlowBursts except for the first one. These two methods yield the same load balance performance. More interesting patterns are elaborated upon in Section III-E. To the best of our knowledge, we are the first work that uses sketches instead of flow/flowlet table to perform L3 load balancing.

C. Data Structure

As shown in Fig. 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 $\mathcal{B}[i].vote$ used to identify FlowBursts. 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 [20]: 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 $Null$.

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;
    $\mathcal{B}[h(f_i)].vote$ += 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
        $\mathcal{B}[h(f_i)].vote$ −= 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 $ECMP\_next\_port(f_i)$;
```

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 appropriate next hop to forward this packet. The insertion workflow of BalanceSketch processes each packet in one pass, and is of $O(1)$ time complexity. We have fully implemented the insertion workflow in the pipeline of programmable switch (with Tofino ASICs) using P4 [77] language, which has 1.2 GHz clock frequency (see Section IV-B).

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 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 resided flow ends.

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 [78]. We will discuss different manipulating choices in Fig. 17.

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 by setting $\mathcal{B}[h(f_i)]$ 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.

Discussion: BalanceSketch makes two approximations: 1) BalanceSketch uses the "vote" field to approximately identify FlowBursts. Recall that in Section II-A, we formally define FlowBurst using speed and size. Although we can use more fields to exactly represent the speed, size, and hash collisions, we find that using just the "vote" field can already achieve high accuracy. Therefore, to save memory, BalanceSketch only use one "vote" field to approximately reflect the speed and size of flowlets. 2) BalanceSketch uses the *follower approximation* strategy to make load balance decisions for FlowBurst followers. BalanceSketch considers subsequent flowlets after crossing the "$\mathcal{F}$" threshold as FlowBursts and manipulates them. We make this approximation because we cannot immediately predict a flowlet as FlowBurst when it just starts. Experimental results show that under these approximations, BalanceSketch still has high accuracy (Section V-A).

Example settings (Fig. 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 30 ms, the *flowlet threshold* δ is 5 ms, and the *voting frequency threshold* $\mathcal{F}$ is 30.

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

Example 2 (center of Fig. 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 Fig. 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

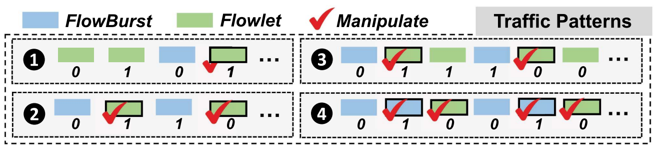


Fig. 3. Examples of typical traffic patterns.

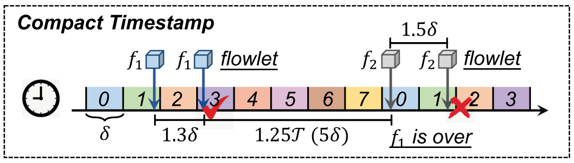


Fig. 4. Compact timestamp ($\Delta = 4\delta$).

f_3. Since $\mathcal{B}[h(f_3)].nexthop = Null$, we forward the packet using ECMP.

E. Handling Different Traffic Patterns

We examine four typical traffic patterns to demonstrate how BalanceSketch manages them, showcasing that our FlowBurst *follower approximation* technique attains comparable load balancing performance to the optimal solution of manipulating each FlowBurst at start. 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. We assume the traffic of each flow consists of high-density FlowBursts and low-density flowlets. An ideal load balance solution should distribute these high-density FlowBursts among all equivalent links as uniformly as possible. Manipulating other flowlets benefits little for load balance as low-density flowlets contribute little to link congestion.

Pattern 1 (upper-left of Fig. 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. In this case, manipulating other flowlets 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 Fig. 3): This pattern consists of FlowBurst1, flowlet1, FlowBurst2, flowlet2, ... 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 Fig. 3): This pattern consists of FlowBurst1, flowlet1, flowlet2, FlowBurst2, flowlet3, flowlet4, ... BalanceSketch manipulates each BurstFollower (e.g., flowlet1), and forwards following flowlet2 and FlowBurst2 through the same next hop. The next hops of BalanceSketch are $\langle 0,1,1,1,0,0,\ldots\rangle$, while that of the ideal solution are $\langle 1,1,1,0,0,0,\ldots\rangle$. 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 Fig. 3): This pattern consists of FlowBurst1, FlowBurst2, flowlet1, FlowBurst3, FlowBurst4, flowlet2, ... BalanceSketch manipulates each latter FlowBurst and each flowlet, and its next hops are $\langle 0,1,1,1,0,0,\ldots\rangle$. It achieves similar performance as the ideal solution with the next hops of $\langle 1,1,0,1,1,0,\ldots\rangle$.

F. BalanceSketch Optimizations

Flow Fingerprint: We use fingerprints (hash values) to replace flow IDs (usually 104 bits) in BalanceSketch, to improve its memory efficiency. In this way, the memory overhead of BalanceSketch is independent to the size of flow ID. Due to hash collisions, some flows could share the same fingerprint, making BalanceSketch regards multiple flows as one flow. Given a flow, the probability that it suffers from fingerprint collisions is $\Pr[collision] = 1-(1-2^{-n})^{\frac{M}{l}}$, where n is the fingerprint size (in bit), M is the number of distinct flows in the network stream, and l is the number of buckets in BalanceSketch. This probability is low, and thus has little impact on performance. For example, when using 16-bit fingerprints and $l = 50,000$ buckets, for $M = 1,000,000$ concurrent flows in the network, the probability of fingerprint collision is just 3.05×10^{-4}, which is negligible. Experiments show that the accuracy of BalanceSketch does not decrease when using 16-bit fingerprints.

Field Combination: Once a FlowBurst is detected, we change the `next_hop` field to manipulate it. After that, even if the `vote` field is decremented to zero, we do not evict this flow in consideration of manipulating less flows (Case 3 in Section III-D). In other words, once `next_hop` is set to a non-$Null$ value, we do not need `vote` field any more. Thus, we can combine `vote` and `next_hop` into one field `vote_hop`.

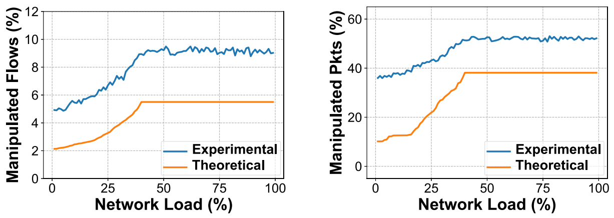
Compact Timestamp: We propose to compress the full `timestamp` into s-bit cell, and use the cell to compactly record the approximate time. Below we take 3-bit cell as an example. As shown in Fig. 4, noticing that 3-bit cell can represent eight states, we cyclically divide the timeline into eight kinds of time slices ($0 \sim 7$), and the length of each time slice is δ. We record these time slices ($0 \sim 7$) rather than the full timestamps in BalanceSketch. If two adjacent packets of a flow are spaced by at least one time slice, we think the second packet is the start of a flowlet. Suppose $\Delta = k\delta$, then if the last arrival time of a flow f_i and the current time are spaced by at least k time slices, we think flow f_i ends. Fig. 4 shows four consecutive packets mapped into the same bucket. The two packets of f_1 are spaced by one time slice, so we report the second packet as the start of a flowlet. Suppose $\Delta = 4\delta$, then since the last packet of f_1 and the first packet of f_2 are spaced by four time slices, when the first packet of f_2 arrives, we think f_1 ends and evict it.

The compact timestamp technique gains memory efficiency at the cost of perceiving time in a fuzzy way. When the interval between two adjacent packets is among $\delta \sim 2\delta$, BalanceSketch might not be able to correctly report the second packet as the start of a flowlet, depending on the relative offset of the timeline. Specifically, only when the interval span three time slices can BalanceSketch report the flowlet correctly. This issue is illustrated in the last two packets in Fig. 4. Although the interval between the two packets of f_2 exceeds δ, BalanceSketch cannot correctly divide them into two flowlets because the interval span just two time slices. Actually, more precision can be attained by using more bits per timestamp or using multiple timestamps with different timeline offsets, but we find that one 8-bit timestamp suffice for good performance in our experiments (see Section V-A1).

Analysis: We derive the error of our Compact Timestamp technique. For an arbitrary flow f, we assume all of its flowlets arrives according to a Poisson process of intensity λ. Let $flowlet_i$ be the i_{th} flowlet of f, and let x_i be the time interval between the arrival time of $flowlet_{i-1}$ and $flowlet_i$. In our derivation, we ignore the timespan of flowlet because it does not affect the final conclusion. Thus, we have $x_i - \delta$ follows an exponential distribution $Exp(\lambda)$. For $flowlet_i$, BalanceSketch cannot correctly report it if the interval between $flowlet_{i-1}$ and $flowlet_i$ span only two time slices, i.e., x_i only spans two slices. For a certain x_i, the probability that it spans two slices is $\frac{2\delta - x_i}{\delta}$. Let A_i be the event that BalanceSketch fails to detect $flowlet_i$. We have $\Pr[A_i] = \int_{\delta}^{2\delta} \lambda e^{-\lambda(x_i-\delta)} \cdot \frac{2\delta - x_i}{\delta} dx_i = \frac{\lambda\delta + e^{-\lambda\delta} - 1}{\lambda\delta}$. Let $\delta' = \frac{1}{\lambda}$ be the average interval between adjacent flowlets, which are usually of RTT scale [2]. In practice, we set δ to be of sub-RTT scale, meaning that $\delta' > \delta$, i.e., $\lambda\delta < 1$. Thus, we have $\Pr[A_i] < \frac{1}{e}$. As stated above, we can also use multiple timestamps with different timeline offsets to further reduce the error rate. Actually, using y timestamps can reduce the error by y times. When $\lambda\delta = 0.1$ and $y = 3$, we have $\Pr[A_i] \approx 1.2 \times 10^{-4}$, which is negligible.

Multi-cell BalanceSketch: To further improve the accuracy, we propose the multi-cell version of BalanceSketch by extending each bucket of BalanceSketch into an array of d (e.g., 4) cells. Each cell consists of four fields: `flow_ID`, `timestamp`, `vote`, and `next_hop`. For each incoming packet of flow f_i at t_{now}, it is mapped into bucket $\mathcal{B}[h(f_i)]$. We first check whether f_i is recorded in a cell in $\mathcal{B}[h(f_i)]$. If so, we increment `vote` and update `timestamp`. Otherwise, we second check whether there is an empty or outdated cell in $\mathcal{B}[h(f_i)]$. If so, we insert f_i into this cell. Otherwise, we third find the cell with the minimum `vote`, and decrement its `vote` by one. If `vote` is decremented to zero and `next_hop` is $Null$, we replace the residing flow in this cell with f_i. Finally, if f_i is recorded in a cell in $\mathcal{B}[h(f_i)]$ and its `next_hop` is not $Null$, we forward the packet through `next_hop`. Otherwise, we forward the packet using ECMP. The insertion workflow of multi-cell BalanceSketch can be accelerated by SIMD instructions [79] on CPU platforms. Experimental results show that using 8-cell buckets can improve the accuracy of BalanceSketch by up to 20%.

Automating Parameter Configuration: As described in Sections II-A and III-D, we use a speed threshold $\mathcal{V}$ to define FlowBurst, and use a voting threshold $\mathcal{F}$ in BalanceSketch to identify FlowBursts. However, we should need different thresholds in different environments. Intuitively, under low network load, there are less risk of load imbalance, and thus we should manipulate less traffic to avoid making the network more chaotic. In such case, we should use large $\mathcal{V}$ and $\mathcal{F}$ to reduce the number of FlowBursts. Whereas under high network load, we should manipulate more traffic to better balance the load, and thus we need smaller $\mathcal{V}$ and $\mathcal{F}$ to increase the number of FlowBursts. On the other hand, these thresholds should not be too small to overwhelm the sketch with excessive FlowBursts. Based on these ideas, we devise a method to automatically setting $\mathcal{V}$ and $\mathcal{F}$ according to current network load (or remaining bandwidth) as follows: $\mathcal{V} \leftarrow \mathcal{V}_0 \times \exp(\frac{-4}{\max(0.6, 1-p)})$, and $\mathcal{F} \leftarrow \mathcal{V} \times \delta$,[4] where $\mathcal{V}_0$ is the maximum bandwidth and p is the current network load. We conduct experiments to evaluate the performance of our automatic parameter configuration method, where we evaluate the ratio of the theoretically/experimentally manipulated flows/packets under different network load. As show in Fig. 5, under low network load, we manipulate a small fraction of traffic. As network load increases, we gradually manipulate more flows/packets to better balance the load. When network load exceeds ξ ($\xi = 40\%$ in our experiments), as network load increases, we no longer manipulate more traffic, so as to avoid overwhelming BalanceSketch with too many FlowBursts. In practice, ξ can be dynamically set according to the scale/skewness of traffic, and the size of BalanceSketch.



(a) Manipulated flows. (b) Manipulated packets.

Fig. 5. Ratio of manipulated flows/packets under automatic parameter configuration.

G. Mathematical Analysis

We mathematically analyze the ability of BalanceSketch to identify FlowBursts, and validate our theoretical results with experiments. To simplify the derivation, we first make the following assumptions for the traffic model.

Assumption III.1. Given a bucket in BalanceSketch, we assume that all active flowlets mapped into this bucket have the same speed, i.e., the interval between any two consecutive packets of any flowlet is constant. Therefore, we can divide the timeline into consecutive time slots of equal length δ_s. Each flow has at most one packet in each time slot. A flowlet is a group of continuous packets that arrive in several consecutive time slots.

Assumption III.2. Given a bucket in BalanceSketch, we assume that the flows mapped into this bucket are: 1) A large flow f, whose flowlet size is always n_1 (in packets), and the interval between its any two consecutive flowlets is always n_2 (in time slots). Note that we define the interval between two flowlets as the time gap between their first packets. 2) Many small flows.

[4]These formulas might not be optimal. We can further consider the size of BalanceSketch and devise smarter formulas. But the results show that these formulas can already well achieve our design goal.

We divide the small flows into flowlets, and assume that these flowlets obey the $M/M/\infty$ queuing theory model: the interval between any two consecutive flowlets obeys the exponential distribution with parameter λ_1, and the size of each flowlet obeys the exponential distribution with parameter λ_2.

We give the well-known conclusion of $M/M/\infty$ model in queuing theory through the following lemma.

Lemma III.1. In each time slot, the number of packets of small flows satisfies Poisson distribution with parameter $\lambda = \frac{\lambda_2}{\lambda_1}$. Let Y_i be the number of packets of small flows arriving in the i^{th} time slot, then $\Pr[Y_i = k] = e^{-\lambda} \cdot \frac{\lambda^k}{k!}$.

To simplify the derivation, we assume Y_i is *independent and identically distributed* for different i, and that the packet belonging to the large flow f is always the last to arrive in each time slot.

Assumption III.3. Given a bucket in BalanceSketch, let X_i be the value of its `vote` field after the i^{th} time slot ($X_0 = 0$). For the large flow f, we assume that it starts at the first time slot. For the other small flows, to simplify the derivation, we assume that the number of packets of small flows in each time slot is *independent and identically distributed*, and according to Lemma 3.1, all obey Poisson distribution with parameter λ. We also assume that the packet belonging to the large flow f is always the last to arrive in each time slot, therefore the `flow_ID` is always f during the first n_1 time slots according to our algorithm.

We can derive the following conclusions about the Markov process of random variable X_i (Lemma 3.2), and the ability of BalanceSketch to detect FlowBurst (Theorem 1).

Lemma III.2. The random variable X_i, i.e., the value of the vote filed satisfies the following Markov process when $i \leqslant n_1$,

$$X_i = \max(X_{i-1} + 1 - Y_i, 1). \tag{1}$$

Further, we can obtain that $1 \leqslant X_i \leqslant i$, and for $\forall i \geq 2$,

$$\Pr[X_i = k] = \begin{cases} \sum_{j=0}^{i-k} \Pr[X_{i-1} = k + j - 1] \cdot e^{-\lambda} \cdot \frac{\lambda^j}{j!} & \\ & 2 \leqslant k \leqslant i \\ 1 - \sum_{k=2}^{i} \Pr[X_i = k] & k = 1 \end{cases}. \tag{2}$$

Theorem 1. Given the flowlet threshold $\delta \leqslant n_2 \cdot \delta_s$, and the voting threshold $\mathcal{F} \leqslant n_1$, where δ_s is the length of the time slot, the probability P_f that the largest flow f is successfully reported as FlowBurst after the $(n_1 + n_2)^{th}$ time slot satisfies

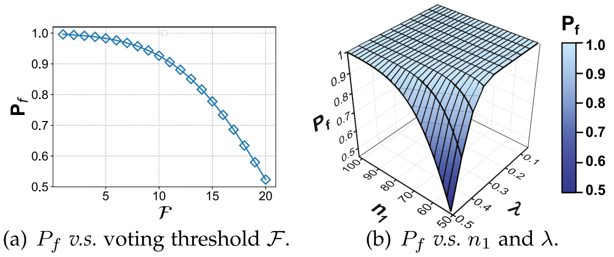
$$P_f \geqslant \Pr\left[X_{n_1} > \mathcal{F} + \sum_{i=1}^{n_2} Y_{n_1+i}\right] \geqslant 1 - \lambda \cdot \frac{n_1 + n_2}{n_1 - \mathcal{F}}. \tag{3}$$

Proof. Based on Lemma 3.2, we have $X_i \geqslant 1 + X_{i-1} - Y_i$, therefore X_{n_1} satisfies

$$X_{n_1} = X_0 + \sum_{i=1}^{n_1}(X_i - X_{i-1}) \geqslant n_1 - \sum_{i=1}^{n_1} Y_i.$$

Then according to the Markov inequality, we have

$$P_f \geqslant \Pr\left[X_{n_1} - \sum_{i=1}^{n_2} Y_{n_1+i} > \mathcal{F}\right]$$



(a) P_f *v.s.* voting threshold $\mathcal{F}$. (b) P_f *v.s.* n_1 and λ.

Fig. 6. Numerical verification of probability P_f with default setting of $n_1 = 70, n_2 = 30, \lambda = 0.5, \mathcal{F} = 10$.

$$\geqslant \Pr\left[n_1 - \sum_{i=1}^{n_1+n_2} Y_i > \mathcal{F}\right] = 1 - \Pr\left[\sum_{i=1}^{n_1+n_2} Y_i \geqslant n_1 - \mathcal{F}\right]$$

$$\geqslant 1 - \frac{\mathbb{E}\left[\sum_{i=1}^{n_1+n_2} Y_i\right]}{n_1 - \mathcal{F}} = 1 - \lambda \cdot \frac{n_1 + n_2}{n_1 - \mathcal{F}}.$$

Experimental analysis (Fig. 6): We conduct experiments to validate our mathematical analyses. Although we cannot directly obtain the analytical solution of P_f from Lemma 3.2, we can give the numerical solution of P_f under a specific setting by numerical simulation. By setting $n_1 = 70, n_2 = 30, \lambda = 0.5, \mathcal{F} = 10$ as default, we show how P_f changes with $\mathcal{F}$ in Fig. 6(a), and how P_f changes with n_1 and λ in Fig. 6(b). The numerical results show that when the flow f is large (i.e., n_1 is large), the number of concurrent small flows is small (i.e., λ is small), and the voting threshold is small (i.e., $\mathcal{F}$ is small), the largest flow f will be correctly reported as FlowBurst with a high probability.

H. *Extension to Network Measurement*

Besides L3 load balancing, our FlowBursts and BalanceSketch can do more to improve the network. In this subsection, we show how to utilize FlowBursts to perform robust per-flow per-hop measurement. We focus on four important tasks: tracing forwarding path, finding the flows consuming huge bandwidth (heavy hitters), finding the flows experiencing packet drops, and finding the flows experiencing inflated queuing delays. The information of these abnormal flows can guide the network operator to quickly locate culprit devices and further debug the network.

In BurstBalancer, we deploy one BalanceSketch on each switch. We configure all BalanceSketches to report the detected FlowBursts along with their attributes (including the timespan, packet count, *etc.*) to control planes. A central analyzer periodically collects the information of FlowBursts from all switches, and further analyze these information to identify abnormal flows. In practice, the clocks of different network devices are hard to be perfectly synchronized [80], [81], which brings errors to exiting measurement systems [48], [82]. Fortunately, we find that by performing network measurement with FlowBursts, BurstBalancer is robust to imperfect clock synchronization. Below we describe how to configure BurstBalancer to perform the four tasks.

Tracing forwarding path: We add two fields to each bucket of BalanceSketch (or each cell in multi-cell BalanceSketch. We will no longer emphasize this point.), which record the start/end Sequence Number of the FlowBurst. We configure each switch to report the Sequence Number range, and the next-hop for each detected FlowBurst. For any FlowBurst, the central analyzer can use its Sequence Number range and per-switch next-hop to track its forwarding path, so as to find the culprit devices where network anomalies happen.

Finding per-hop heavy hitters: We add a packet count field to each bucket, and let each switch report the packet count for each FlowBurst. For each active flow[5] f, the analyzer acquires its per-hop size by summing up the packet counts of all FlowBursts in f. In this way, the analyzer can find those flows that are consuming a large amount of bandwidth on each switch, so as to decide whether to impose some restrictions on them (e.g., reducing their priorities).

Locating packet drops: We add three fields to each bucket: a packet count field, two fields recording the start/end Sequence Number of the FlowBurst. We configure each switch to report the packet count, the Sequence Number range, and the next-hop for each FlowBurst. As stated above, the central analyzer uses the Sequence Number range and per-switch next-hop to track each FlowBurst along its forwarding path. For each FlowBurst, the analyzer can locate those culprit switches where its packet count decreases, so as to further troubleshoot the network.

Locating inflated queuing delays: We add four fields to each bucket: two timestamp field recording the start/end time of the FlowBurst, and two fields recording the Sequence Number range of the FlowBurst. We configure each switch to report the timespan, the Sequence Number range, and the next-hop for each FlowBurst. The central analyzer uses Sequence Number range and next-hop to trace each FlowBurst, and finds those culprit switches where the FlowBurst timespan suddenly increases.

Discussion: As a network measurement system, BurstBalancer has three advantages. 1) Fine-grained: BurstBalancer can acquire per-flow per-hop information, and thus can perform network-wide measurement tasks; 2) Lightweight: Unlike many systems that collect packet-level information for all flows [43], [82], [83], BurstBalancer only records a small fraction of critical flows, and these flows are important to network performance. Thus, BurstBalancer significantly reduces the memory and bandwidth overhead; 3) Robust: BurstBalancer uses FlowBurst to perform network measurement, which is robust to imperfect clock synchronization. In conclusion, besides L3 load balancing, another promising direction is to use FlowBursts to perform accurate and lightweight network measurement.

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 Fig. 7, we deploy one BalanceSketch on each edge switch and let it process all packets arriving from the line side. Given

[5]We define a flow as an active flow if the time interval between the arrival time of its last packet t_{last} and the current time t_{now} is smaller than the flow timeout threshold Δ, namely $|t_{now} - t_{last}| < \Delta$.

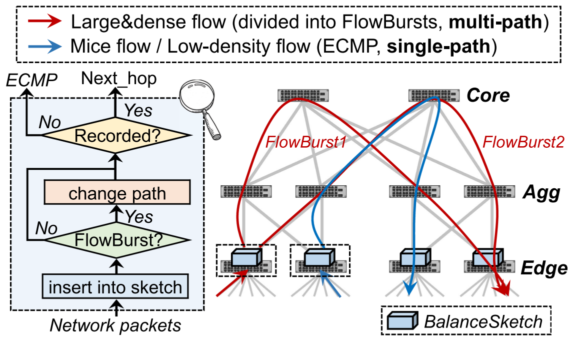


Fig. 7. BurstBalancer overview.

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 fully implement BurstBalancer on a testbed with 4 Edgecore Wedge 100BF-32X switches (with Tofino ASIC) [55] and 16 end-hosts in a Leaf-Spine topology. On each switch, we develop BalanceSketch using P4 language [77].

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, *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 $t_{now} - \Delta$), BalanceSketch needs to clear the bucket by setting `flow_ID` to *Null* (Case 1 in Section 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. Here, the mirror and recirculate mechanism would not cause performance issue. First, only a few packets (<0.5%) need this mechanism. Second, this mechanism is only used to clear the outdated bucket, which would not affect the scheduling and forwarding of packets.

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 Section 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 our 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.

Hardware constraints: In Section III-F, we propose a field combination technique to combine `vote` and `next_hop` fields into one `vote_hop` field. However, if we combine these two fields in our hardware implementation, there will be three fields that have pairwise dependencies on each other: `flow_ID`, `timestamp`, and `vote_hop`. For example, for each incoming packet, we first check `flow_ID` field, and then update `timestamp` and `vote_hop` field. After checking `timestamp` and `vote_hop`, we decide whether to evict the residing flow by changing `flow_ID` field. Unfortunately, P4 [77] only supports simultaneously accessing at most two variables. Thus, we need a kind of redundant design to resolve the mutual dependencies of the three fields, which is done by creating a duplicate for each field in our implementation.

2) Workflow: As shown in Fig. 8, 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 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 Δ. 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.

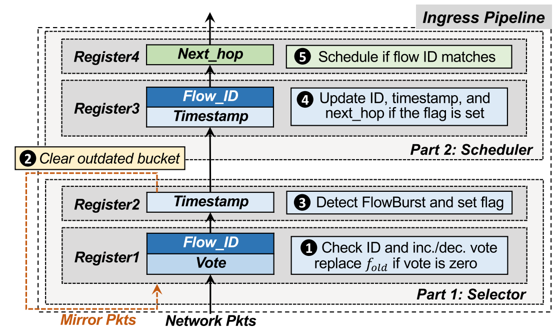

Ingress Pipeline
Register4
Next_hop
5 Schedule if flow ID matches
Register3
Flow_ID
Timestamp
4 Update ID, timestamp, and next_hop if the flag is set
Part 2: Scheduler
2 Clear outdated bucket
Register2
Timestamp
3 Detect FlowBurst and set flag
Register1
Flow_ID
Vote
1 Check ID and inc./dec. vote replace f_{old} if vote is zero
Part 1: Selector
Mirror Pkts
Network Pkts

Fig. 8. BalanceSketch on programmable switch.

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%

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.

C. Discussion

BurstBalancer differentiates itself by manipulating only a small fraction of flows. This aspect enables it to seamlessly integrate with numerous network measurement and management systems, including but not limited to 007 [37] and HPCC [42]. To illustrate, let's consider the 007 system, which assumes all flows follow ECMP. After detecting TCP retransmission on end-hosts, 007 triggers a path discovery mechanism to acquire the routing links of the victim flow. Subsequently, it employs a voting scheme based on the paths of flows that had retransmissions, and the top-voted links are reported in each measurement epoch. In LetFlow, all flows have unfixed forwarding path, which changes rapidly and randomly, making the path tracing scheme impossible to implement. By contrast, in BurstBalancer, most flows follow ECMP and thus have fixed forwarding paths. In BurstBalancer, if a TCP retransmission is detected for a ECMP flow, 007 can still trace its forwarding path and update the votes for each link along the path. The voting outcomes can then accurately reflect the real-time congestion level of each link.

Thus, BurstBalancer maintains its compatibility with systems like 007.

V. Experimental Results

We extensively evaluate BurstBalancer **(BB)** with CPU experiments (Section V-A), large-scale simulations (Section V-B), and testbed experiments (Section V-C). 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 $> 90\%$ Recall Rate in finding FlowBursts. (Section 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 [2], BurstBalancer manipulates 58 times less flows while better balances the traffic. (Section V-A2)
- *In symmetric topologies, can BurstBalancer better balance the traffic?* We extensively evaluate BurstBalancer using simulations. As a whole, BurstBalancer achieves 5%$\sim$35% better FCT than state-of-the-art LetFlow [2] and DRILL [14] in symmetric topologies. (Section V-B)
- *In asymmetric topologies, can BurstBalancer better balance the traffic?* We evaluate BurstBalancer on a small-scale testbed with asymmetry. The results show that BurstBalancer achieves up to 30$\times$ better FCT than LetFlow and up to 6.4$\times$ better FCT than WCMP [21]. (Section V-C)
- *Can BurstBalancer be well deployed into commercial switches?* We evaluate BurstBalancer on an electronic system level (ESL) simulation platform. The cycle-level results show that BalanceSketch can be well deployed into commercial chips and BurstBalancer achieves good load balancing performance in RDMA networks. (Section V-B2)

Metrics: We use flow completion time (FCT) as the primary metric. We also consider the statistics of the queue lengths across ports and the packet reordering ratio. We use the Recall Rate (RR) and Average Relative Error (ARE) to evaluate the accuracy of BalanceSketch, which are defined as follows.

1) Recall Rate (RR): Ratio of the number of correctly reported instances to the number of ground-truth instances.

2) Average Relative Error (ARE): $\frac{1}{|\boldsymbol{\Psi}|}\sum_{f_i\in\boldsymbol{\Psi}}|n_i-\hat{n_i}|/n_i$, where n_i is the real size/timespan of FlowBurst f_i, $\widehat{n_i}$ is its estimated size/timespan, and $\boldsymbol{\Psi}$ is the set of all FlowBursts.

Workloads: We use three realistic workloads and one synthetic workload: 1) Web search workload [84] from a production cluster running web search services, where the average flow size is $\sim 2.5\times10^6$ bytes; 2) RPC workload [85] that contains many small flows, where the average flow size is $\sim 2\times10^2$ bytes; 3) Enterprise workload that is derived from our Huawei data center running Hadoop applications, where the average flow size is $\sim 5.7\times10^4$ bytes. 4) Synthetic workload that is of heavy-tailed distribution, where the average flow size is ~ 30 packets. The traffic distribution is shown in Fig. 9. All the four workloads are heavy-tailed: a small fraction of large flows contribute to most traffic.

Parameter selection: We set the parameters of BurstBalancer intuitively: 1) We set the flowlet timeout threshold δ to a sub-RTT timescale. As suggested in LetFlow [2], δ controls the trade-off between load balance and packet reordering. Larger δ goes with fewer reordering packets and greater risk of load imbalance. A sub-RTT timescale δ can well divide TCP bursts into flowlets and achieve good performance. 2) We set the flow timeout threshold Δ to a RTT timescale. BalanceSketch uses Δ to identify whether a residing flow ends, so we set Δ to 3$\sim$5 times of RTT. 3) We set the voting threshold $\mathcal{F}$ to a small value, because we find that the BalanceSketch using small $\mathcal{F}$ can accurately detect FlowBursts.

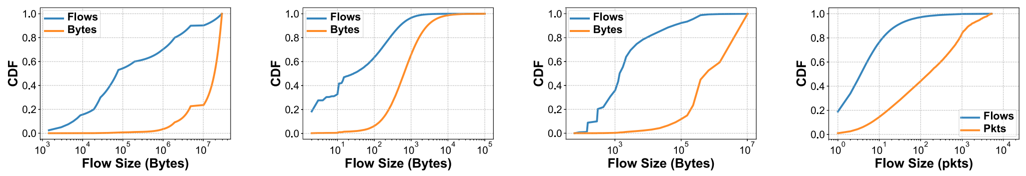
A. CPU Experiments

1) Performance of BalanceSketch: We evaluate the performance of BalanceSketch under small memory usage. As we are the first to propose the concept of FlowBurst, and considering that there are no existing works that can be directly used to find FlowBursts, we implement the strawman solution described in Section III-A as the baseline approach. We implement basic BalanceSketch, and the optimized BalanceSketch using 16-bit flow ID fingerprints and 8-bit compact timestamps.

Dataset: We use the IMC packet traces [54] collected in a data center network, which contains about 19.9 M packets belonging to 7.6 M different flows. We set the flowlet threshold $\delta = 50\mu s$, set $\mathcal{V}$ to the 70th percentile of the speed of all active flowlets, and set η_k to the size of the 200th largest flowlets with $> \mathcal{V}$ speed. In other words, we define the top-200 largest flowlets with $> \mathcal{V}$ speed as FlowBursts.

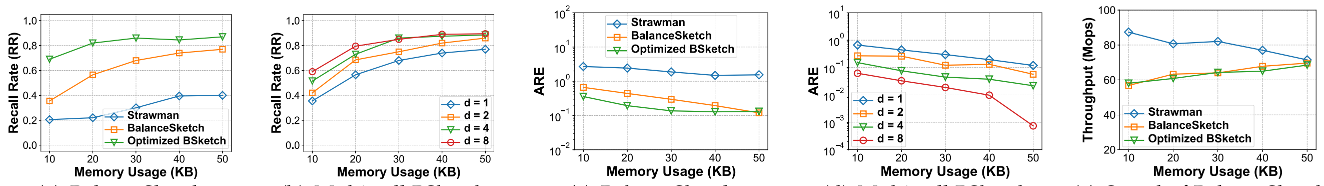
Accuracy of basic BalanceSketch (Fig. 10(a)): We find that the RR of BalanceSketch greatly outperforms the strawman solution, and the RR of the optimized BalanceSketch is higher than basic BalanceSketch. Compared to the strawman solution, RR of BalanceSketch is about 20% higher on average. The optimized BalanceSketch improves RR by about $10\% \sim 33\%$ compared to the basic version.

Accuracy of multi-cell BalanceSketch (Fig. 10(b)): We find that for multi-cell BalanceSketch, larger d goes with higher RR. Compared to the basic BalanceSketch, the multi-cell BalanceSketch with $d = 8$ improves RR by about 20% on average. The results show that when using 50 KB of memory, the multi-cell BalanceSketch with $d = 8$ achieves RR of 90%.



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

Fig. 9. Traffic distributions. The Bytes (Pkts) CDF shows the distribution of traffic bytes (packets) across flow sizes.



(a) BalanceSketch. (b) Multi-cell BSketch. (c) BalanceSketch. (d) Multi-cell BSketch. (e) Speed of BalanceSketch.

Fig. 10. Performance of BalanceSketch (BSketch) and optimized BalanceSketch.

Accuracy of FlowBurst size estimation (Fig. 10(c)–(d)): We find that the ARE of BalanceSketch is one order of magnitude lower than the strawman solution, and the ARE of the optimized BalanceSketch is lower than basic BalanceSketch. When using 40 KB of memory, ARE of the strawman solution, the basic BalanceSketch, and the optimized BalanceSketch are 1.51, 0.20, and 0.13, respectively. *We also find that for multi-cell BalanceSketch, larger d goes with lower ARE.* Compared to the basic BalanceSketch, the multi-cell BalanceSketch with $d = 8$ improves ARE by $1 \sim 2$ orders of magnitudes on average. When using 50 KB of memory, the multi-cell BalanceSketch with $d = 8$ achieves ARE of 7.6×10^{-4}.

Speed of BalanceSketch (Fig. 10(e)): We find that on CPU platform, our BalanceSketch achieves >60 Million operations per seconds processing speed, which is faster than most sketch algorithms [75]. In our hardware implementation (Section V-C), we deploy BalanceSketch into the pipeline of the switch ASICs, whose speed is only affected by the clock frequency. For example, the Tofino switches used in our testbed have 1.2 GHz clock frequency.

Analysis: We find that BalanceSketch greatly outperforms the strawman solution. The results are consistent with our analysis in Section 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 unnecessary flowlets, gaining high memory efficiency. *We also find that optimized/multi-cell BalanceSketch is more efficient.* The results show that 16-bit flow fingerprint and 8-bit compact timestamp are sufficient for good performance. *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 [20] and LetFlow [2]. We use C++ to simulate the load balancing module of a 128-port switch, on which we deploy the Flowlet Tables (LetFlow) and the BalanceSketchs with different sizes (2 K/4 K # entries/buckets). In our setting, there are two switches connected by 128 links. We generate the traffic according to the synthetic workload (Fig. 9(d)) at switch 1. We measure the traffic distribution across the 128 links, and count the reordering packets at switch 2.

Load distribution across all ports (Fig. 11(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 2 K buckets is smaller than LetFlow using 4 K 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 unbalanced load.

Ratio of manipulated flows (Fig. 11(b)): We find that compared to LetFlow, BurstBalancer manipulates 58× fewer flows while better balance the load. The results show that the manipulated flows of BurstBalancer is 1.0 %~1.65%, while that of LetFlow is $> 95\%$. Note the the load balance performance of BurstBalancer_2 K is better than LetFlow_4 K.

Ratio of reordering packets (Fig. 11(c)): We find that compared to LetFlow using 4 K entries, BurstBalancer using 2 K buckets has less reordering packets while achieves better load balance performance. We simulated a scenario where two switches S_1 and S_2 are connected by 128 links. We generate traffic at S_1, and measure the packet reordering rate at S_2 by counting the mismatches between actual and expected sequence number.

Load distribution for high-density traffic (Fig. 11(d)): To better demonstrate the advantages of our BurstBalancer over LetFlow, we accelerate the synthetic workload by 5 times to create a high-density traffic model. We repeat the experiments using LetFlow_4 K and BurstBalancer_2 K. 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.

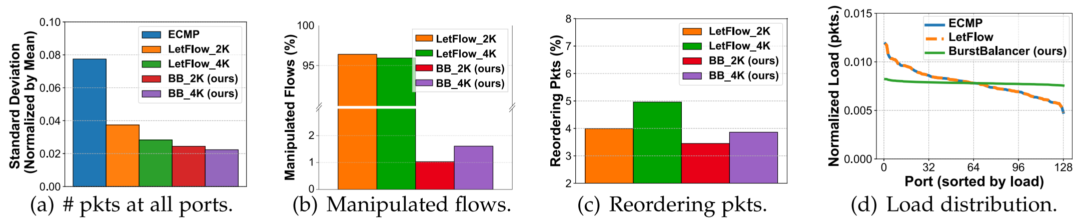


Fig. 11. Performance of BalanceSketch on single switch.

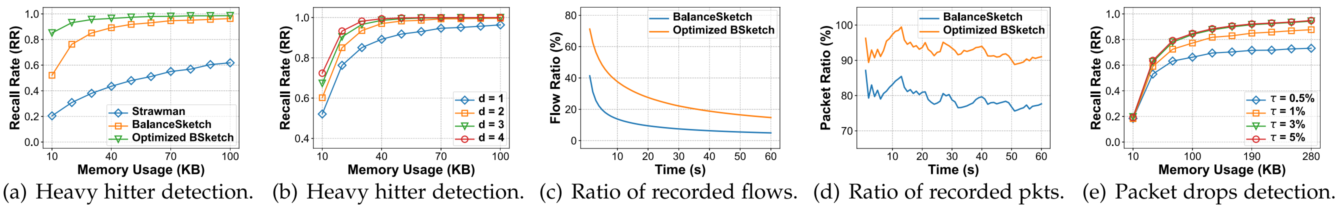


Fig. 12. Performance on finding high-bandwidth-consuming flows (a-d) and detecting packet drops (e).

BurstBalancer can still well balance the traffic since it only manipulates critical flowlets and ignores abundant unnecessary flowlets.

3) Performance on Network Measurement: We evaluate the performance of BurstBalancer on performing the measurement tasks in Section V-A3. We conduct experiments using CAIDA [86] dataset. For the experiments of locating packet drops and inflated queuing delay, we randomly choose τ packets and let the switch proactively drops these packets or increases their queuing delays.

Accuracy on finding per-hop heavy hitters (Fig. 12(a)–(d)): We find that BalanceSketch can accurately finding heavy hitters with small memory usage. As shown in Fig. 12(a), when using 50 KB memory, BalanceSketch achieves >95% Recall Rate in finding top-200 heavy hitters, which is significantly higher than that of the strawman solution. Fig. 12(b) shows the performance of multi-cell BalanceSketch. We can see that larger d goes with higher accuracy for heavy hitter detection. We fix the memory usage to 300 KB, and illustrate the ratio of flows/packets recorded in BalanceSketch (i.e., the packets/flows that are manipulated by BurstBalancer) in Fig. 12(c)–(d). We can see that BalanceSketch only records a small fraction of flows ($<$ 20%), and these flows contribute to a large amount of traffic ($>$ 80%), meaning that BalanceSketch can effectively finding those high-bandwidth-consuming flows.

Accuracy on detecting packet drops (Fig. 12(e)): We find that BurstBalancer achieves up to 97% Recall Rate in detecting packet drops. We can see that larger packet drop rate goes with higher accuracy, and even under $\tau = 0.5\%$ packet drop rate, BurstBalancer can still achieve up to 73% Recall Rate to detect packet drops.

Accuracy on detecting inflated queuing delays (Fig. 13): We find that BurstBalancer achieves up to 94% Recall Rate in detecting inflated queuing delays. From Fig. 13(a), we can see

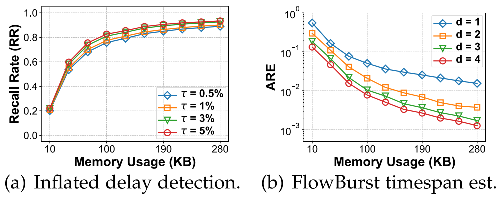


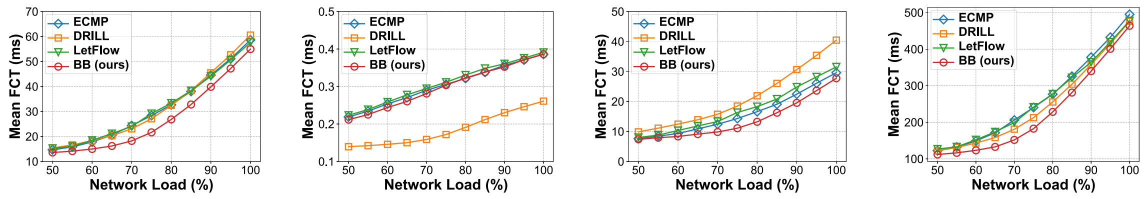
Fig. 13. Performance on detecting inflated delays.

that as the inflated delay rate decreases, the accuracy of BurstBalancer only slightly decreases. Under $\tau = 0.5\%$ delay rate, BurstBalancer can still achieve up to 90% Recall Rate to detect inflated delays. We also evaluate the accuracy of BurstBalancer on estimating the timespans of FlowBursts in Fig. 13(b). We can see that BalanceSketch achieves up to 10^{-3} ARE, and the basic BalanceSketch can still achieves nearly 10^{-2} ARE, which is very accurate.

B. Simulations

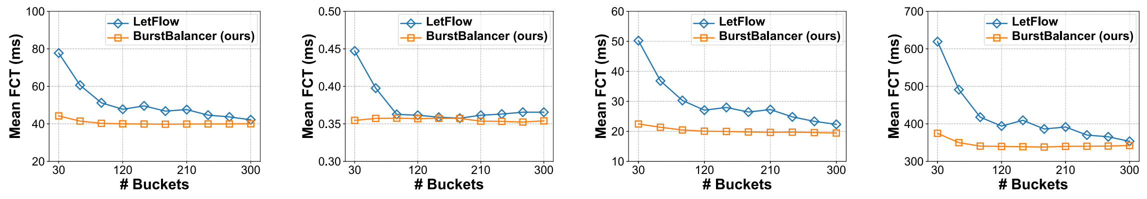
1) Event-Level Simulations (NS-2): We evaluate BurstBalancer using an event-level network simulator, Network Simulator 2 (NS-2) [56], in large-scale symmetric topologies, where we compare BurstBalancer against ECMP [20], DRILL [14], and LetFlow [2] under different network loads. We also evaluate the performance of BurstBalancer and LetFlow using tables of different sizes, validating the memory efficiency of BurstBalancer.

Topology and traffic: We conduct experiments in a two-tier Leaf-Spine topology with 8 spine and 8 leaf switches. Each leaf switch is connected to 16 servers. All links run at 10 Gbps. Here, we have a convergence rate of 2 at the leaf level, which is common in modern data centers [2], [3]. We configure 90% of the bandwidth to deliver the web search workload (Fig. 9(a)), and



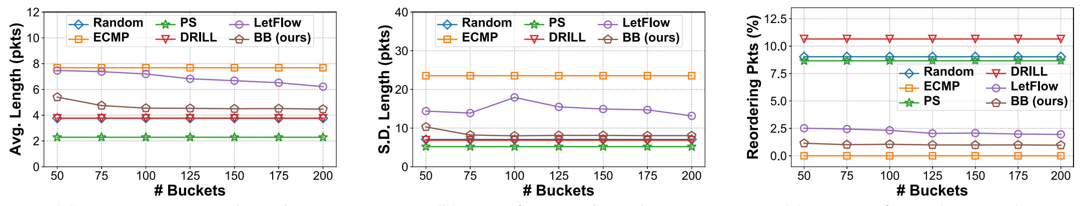
(a) All flows. (b) Small flows (<0.1MB). (c) Medium flows. (d) Large flows (>10MB).

Fig. 14. NS-2 simulation results: FCT statistics under different network loads.



(a) All flows. (b) Small flows (<0.1MB). (c) Medium flows. (d) Large flows (>10MB).

Fig. 15. NS-2 simulation results: FCT statistics of LetFlow and BurstBalancer using tables of different sizes.



(a) Average queue length. (b) SD of queue length. (c) Ratio of reordering pkts.

Fig. 16. HDCN simulation results: queue length statistics and ratio of reordering packets.

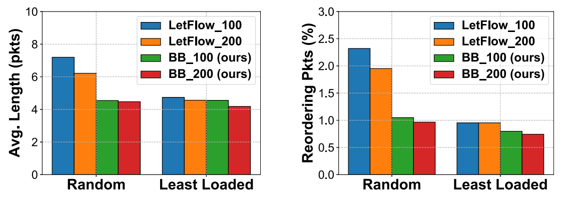
the rest to deliver the RPC workload (Fig. 9(b)) as background traffic.

Setting: For BurstBalancer and LetFlow, we configure the BalanceSketch/Flowlet Table to have 250 buckets/entries by default. In practice, such a small table can fit into one single 1R1W on-chip memory bank, and consumes negligible die area. We set the flowlet threshold $\delta = 200\mu s$, set the flow timeout threshold $\Delta = 50\,ms$, and set $\mathcal{F} = 0$.

FCT versus. network load (Fig. 14): We find that the overall average FCT of BurstBalancer is always lower than ECMP, DRILL, and LetFlow under different network loads. As shown in Fig. 14(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∼10 MB), and large flows ($>$ 10MB) in Fig. 14(b)–(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 versus. number of buckets/entries (Fig. 15): 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 Fig. 15(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 average FCT than LetFlow. We further study the average FCT of flows of different sizes in Fig. 15(b)–(d), and the results are similar to Fig. 15(a).

Analysis: BurstBalancer has lower FCT than LetFlow when using the flowlet tables of the same sizes. When the amount of storage is sufficient, BurstBalancer has similar performance as LetFlow. When the amount of storage is small, LetFlow has poor load balance performance but BursstBalaner can still well balance the traffic. This is because when the number of concurrent flows exceeds the size of the flowlet table, LetFlow inevitably regards multiple flows as one, making it difficult to



(a) Average queue length. (b) Ratio of reordering pkts.

Fig. 17. Performance under different manipulating choices.

detect flowlets. And thus, LetFlow cannot well balance the traffic when using small flowlet tables. By contrast, BurstBalancer only manipulates a small amount of critical flowlets, which is memory efficient.

2) ESL Simulations (HDCN): We evaluate BurstBalancer on an electronic system level (ESL) network simulator named Hyper Data Center Network (HDCN), where we build a large-scale RDMA network to observe how well can BurstBalancer balance the load, and to what extent is the reordering ratio. We compare BurstBalancer against ECMP [20], per-packet random routing (Random), per-packet round-robin routing (Packet-Spray, PS), LetFlow [2], and DRILL [14].

Platform and implementation: HDCN is an electronic system level (ESL) simulation platform used by Huawei for years. Developed based on SystemC kernel, HDCN offers cycle-level simulation capability, which cannot be achieved by event-level simulators [56], [87], [88]. HDCN integrates general switch models and general NIC models, supports various network topologies (FatTree, VL2, *etc.*), and allows users to customize network configurations. It also offers multiple congestion control algorithms (DCQCN/ECN/PFC). In addition, HDCN utilizes MPI to support distributed/multi-threaded parallel acceleration, achieving simulation speeds significantly higher than current open-sourced simulators like NS2 [56], NS3 [87] and NSPY [88]. Currently, HDCN is extensively used within Huawei for the research and development of novel network-wide algorithms and chip architectures [89]. We implement BalanceSketch by extending a generalized chip model with small modifications of its pipeline processing logic. We set the number of buckets of BalanceSketch to $50 \sim 200$, which can fit into a single 1R1W on-chip memory bank. Thanks to the chip-level visibility of HDCN to any network device, we can directly observe the queue lengths across all fabric ports in our experiments, and use their average or standard deviation to reflect the load balancing performance.

Topology, traffic, and setting: We conduct experiments in a VL2 topology with 4 core, 8 aggregation, and 8 edge switches in 4 pods. The bandwidth of all fabric links are 100 Gbps. Each edge switch is connected to 16 servers through 50 Gbps links. Each server runs Remote Direct Memory Access (RDMA) [90] transport logic in network interface card (NIC). To better observe the distribution of traffic on multi-paths, we disable the go-back-N mechanism of RDMA. We use the enterprise workload (Fig. 9(c)) and configure each client in a pod to send ON-OFF traffic to servers in the other pod, so that all traffic traverses the fabric. The ratio of the ON/OFF duration is $1:5$. The experiments are conducted under $\sim$75% network loads. We set δ to about $1.5\times$ RTT, set $\Delta = 10\delta$, and set $\mathcal{F} = 15$.

Statistics of the queue lengths (Fig. 16(a)–(b)): We find that BurstBalancer better balances the traffic than ECMP and LetFlow, and achieves similar load balance performance as DRILL and per-packet random routing. As shown in Fig. 16(a), the average queue lengths of BurstBalancer are smaller than ECMP and LetFlow, and similar to DRILL and per-packet random routing. Per-packet round-robin has the smallest average queue length. Fig. 16(b) shows the standard deviation of the average queue length across all fabric ports. The results are similar to that of the average queue lengths.

Ratio of reordering packets (Fig. 16(c)): We find that BurstBalancer achieves the lowest packet reordering ratio ($<$ 1%) among candidate schemes (except ECMP). We can see that packet-level schemes (DRILL, per-packet random, and per-packet round-robin) have the highest reordering rate. LetFlow has higher reordering rate than BurstBalancer due to the large difference in path latency caused by unbalanced load. Note that the results in our experiments are worse than that in real scenes because of the high-density workload.

Performance under different manipulating choices (Fig. 17): We find that in symmetric topologies, for BurstBalancer, randomly picking a port for each FlowBurst and picking the least loaded port have similar performance. This is because BurstBalancer balances the traffic so well that there is little room for improvement. For LetFlow, the difference between the two manipulating choices is pronounced.

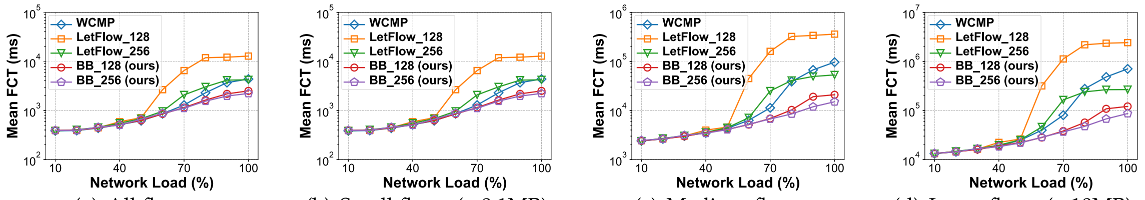
Analysis: Through the cycle-level results in our ESL simulations, we can see that BurstBalancer well balances the traffic with small reordering ratio in RDMA networks. Our ESL experiment also validates that BurstBalancer can be well deployed into commercial chips. The results have been acknowledged by the committee of Huawei, and we have deployed BurstBalancer into our commercial chips as a load balancing function.

C. Testbed Experiments

As described in Section IV-B, we build a small-scale testbed in an asymmetric topology, on which we compare BurstBalancer against WCMP [21], and LetFlow [2].

Topology and traffic: As shown in Fig. 19, we use a two-tier Leaf-Spine topology consisting of 2 spine switches and 2 leaf switches, each of which is connected to 8 servers.

All links run at 40 Gbps. 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 [91], 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 (Fig. 9(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. We configure the bandwidth usage of the single-packet traffic as $\sim$ 5Gbps.



(a) All flows. (b) Small flows (<0.1MB). (c) Medium flows. (d) Large flows (>10MB).

Fig. 18. Testbed results: FCT statistics under different network loads in asymmetric topology.

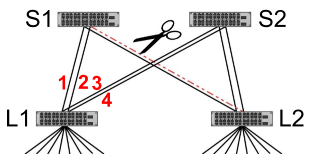
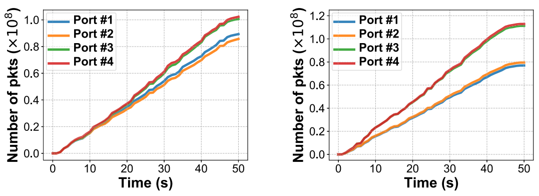


Fig. 19. Testbed topology.



(a) # pkts (ECMP). (b) # pkts (BurstBalancer).

Fig. 20. Testbed results: Number of forwarded pkts.

Setting: For BurstBalancer and LetFlow, we configure BalanceSketch/Flowlet Table to have 128 or 256 buckets/entries. 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 $\Delta = 50\,ms$, and set the voting threshold $\mathcal{F} = 0$.

FCT versus. network load (Fig. 18): 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 Fig. 18(a), 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. 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 uneven, the load balance performance is bad. We further study the average FCT of flows of different sizes in Fig. 18(b)–(d). The results are similar to that in Fig. 18(a).

Forwarding statistics of the four ports in a leaf switch (Fig. 20):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 Fig. 19) 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 Fig. 20(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 Fig. 20(b), for BurstBalancer, the traffic distribution ratio is 1:1.03:1.45:1.47. As explained in LetFlow [2], flowlet switching schemes have the implicit feedback mechanism: once a flow is routed through a congested link, this flow is more likely to experience a flowlet timeout, and thus it is more likely to be rerouted through other links. The results show that BurstBalancer also keeps this implicit feedback mechanism, and achieves the traffic distribution closer to the optimal ratio.

D. Discussion

In our experiments, we juxtaposed the load balance performance of BurstBalancer with LetFlow under the same flowlet table sizes. With sufficient memory allocated for the flowlet table, the load balance performance of BurstBalancer and LetFlow appear to be similar. However, it is worth noticing that BurstBalancer only manipulates a small fraction of flows (<2%), whereas LetFlow manipulates almost all flows (>98%) (Fig. 11(b)). The forwarding paths of most flows in BurstBalancer are fixed and predictable. As a result, BurstBalancer experiences less packet reordering, thereby simplifying network measurement and management. On the other hand, under conditions of limited available memory for the flowlet table, BurstBalancer outperforms LetFlow in load balance performance. Considering the current trend of switch bandwidth growing much faster than the on-chip SRAM, we project that BurstBalancer's efficient memory usage will become increasingly valuable for future networks.

VI. Conclusion

This paper presents BurstBalancer, an efficient load balancing system for data center networks. The design philosophy of BurstBalancer is to only manipulate a small amount of critical flowlets, which are formally defined 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. Our ESL platform (HDCN) has verified the effectiveness and efficiency of BurstBalancer on commercial chips. The results have been acknowledged by the committee of Huawei, and we have deployed BurstBalancer into our commercial chips as a load balancing function.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable suggestions.

References

[1] Z. Liu et al., "BurstBalancer: Do less, better balance for large-scale data center traffic," in *Proc. IEEE 30th Int. Conf. Netw. Protoc.*, 2022, pp. 1–13.

[2] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 407–420.

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

[4] X. Gao, L. Kong, W. Li, W. Liang, Y. Chen, and G. Chen, "Traffic load balancing schemes for devolved controllers in mega data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, pp. 572–585, Feb. 2017.

[5] N. Liu, A. Haider, D. Jin, and X.-H. Sun, "Modeling and simulation of extreme-scale fat-tree networks for HPC systems and data centers," *ACM Trans. Model. Comput. Simul.*, vol. 27, no. 2, pp. 1–23, 2017.

[6] N. Liu, A. Haider, X.-H. Sun, and D. Jin, "FatTreeSim: Modeling large-scale fat-tree networks for HPC systems and data centers using parallel and discrete event simulation," in *Proc. 3rd ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2015, pp. 199–210.

[7] D. Wang, X.-H. Sun, N. Hu, and N. Sun, "EthSpeeder: A high-performance scalable fault-tolerant ethernet network architecture for data center," in *Proc. IEEE 6th Int. Conf. Netw. Architecture Storage*, 2011, pp. 355–363.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.

[9] A. Greenberg et al., "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 51–62.

[10] P. Gratz, B. Grot, and S. W. Keckler, "Regional congestion awareness for load balance in networks-on-chip," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Architecture*, 2008, pp. 203–214.

[11] O. Lysne, T. Skeie, S.-A. Reinemo, and I. Theiss, "Layered routing in irregular networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 1, pp. 51–65, Jan. 2006.

[12] 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 *Proc. IEEE 6th Int. Conf. Cloud Netw.*, 2017, pp. 1–6.

[13] A. K. Y. Cheung and H.-A. Jacobsen, "Dynamic load balancing in distributed content-based publish/subscribe," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2006, pp. 141–161.

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

[15] M. Handley et al., "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 29–42.

[16] Y. Lu et al., "Multi-path transport for RDMA in datacenters," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 357–371.

[17] J. Cao et al., "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol.*, 2013, pp. 49–60.

[18] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2012, pp. 139–150.

[19] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized "zero-queue" datacenter network," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2014, pp. 307–318.

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

[21] J. Zhou et al., "WCMP: Weighted cost multipathing for improved fairness in data centers," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.

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

[23] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 191–205.

[24] F. De Pellegrini, L. Maggi, A. Massaro, D. Saucez, J. Leguay, and E. Altman, "Blind, adaptive and robust flow segmentation in datacenters," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 10–18.

[25] M. Al-Fares et al., "Hedera: Dynamic flow scheduling for data center networks," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implementation*, San Jose, USA, 2010, pp. 89–92.

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

[27] C. Guo et al., "RDMA over commodity ethernet at scale," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 202–215.

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

[29] F. Fan, B. Hu, and K. L. Yeung, "Routing in black box: Modularized load balancing for multipath data center networks," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1639–1647.

[30] Y. Li et al., "DumbNet: A smart data center network fabric with dumb switches," in *Proc. 13th EuroSys Conf.*, 2018, Art. no. 9.

[31] M. Alizadeh et al., "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2014, pp. 503–514.

[32] 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 *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 155–161.

[33] K. He et al., "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 465–478, 2015.

[34] S. Sinha, S. Kandula, and D. Katabi, "Harnessing tcp's burstiness with flowlet switching," in *Proc. 3rd ACM Workshop Hot Topics Netw.*, 2004.

[35] S. Prabhavat, H. Nishiyama, N. Ansari, and N. Kato, "Effective delay-controlled load distribution over multipath networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 10, pp. 1730–1741, Oct. 2011.

[36] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 129–143.

[37] B. Arzani et al., "007: Democratically finding the cause of packet drops," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 419–435.

[38] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao, "Correlation-based traffic analysis attacks on anonymity networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 7, pp. 954–967, Jul. 2010.

[39] H. Huang, S. Guo, P. Li, W. Liang, and A. Y. Zomaya, "Cost minimization for rule caching in software defined networking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 1007–1016, Apr. 2016.

[40] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. Internet Meas. Conf.*, 2017, pp. 78–85.

[41] B. Schlinker et al., "Engineering egress with edge fabric: Steering oceans of content to the world," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 418–431.

[42] Y. Li et al., "HPCC: High precision congestion control," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2019, pp. 44–58.

[43] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "PINT: Probabilistic in-band network telemetry," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 662–680.
[44] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Googles network infrastructure," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 58–72.
[45] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 615–626.
[46] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing packet trajectory in software-defined datacenter networks," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, 2015, pp. 1–7.
[47] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, "Taking the blame game out of data centers operations with NetPoirot," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2016, pp. 440–453.
[48] W. Wang, X. C. Wu, P. Tammana, A. Chen, and T. E. Ng, "Closed-loop network performance monitoring and diagnosis with SpiderMon," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2022, pp. 267–285.
[49] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with PathDump," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 233–248.
[50] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with SwitchPointer," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 453–456.
[51] A. Eswaradass, X.-H. Sun, and M. Wu, "Network bandwidth predictor (NBP): A system for online network performance forecasting," in *Proc. 6th IEEE Int. Symp. Cluster Comput. Grid*, 2006, pp. 4 pp.–268.
[52] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas.*, 2009, pp. 202–208.
[53] M. Alizadeh et al., "pFabric: Minimal near-optimal datacenter transport," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2013, pp. 435–446.
[54] Data Set for IMC 2010 Data Center Measurement. [Online]. Available: https://pages.cs.wisc.edu/tbenson/IMC10_Data.html
[55] Barefoot tofino: World's fastest p4-programmable ethernet switch asics. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/
[56] Network Simulator (ns-2). [Online]. Available: https://www.isi.edu/nsnam/ns/
[57] The source codes related to burstbalancer. [Online]. Available: https://github.com/BurstBalancer/Burst-Balancer
[58] A. Kabbani and M. Sharif, "Flier: Flow-level congestion-aware routing for direct-connect data centers," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[59] P. Wang, G. Trimponias, H. Xu, and Y. Geng, "Luopan: Sampling-based load balancing in data center networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 133–145, Jan. 2019.
[60] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 253–266.
[61] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 266–277, 2011.
[62] Z. Li, J. Bi, Y. Zhang, A. B. Dogar, and C. Qin, "VMS: Traffic balancing based on virtual switches in datacenter networks," in *Proc. IEEE 25th Int. Conf. Netw. Protoc.*, 2017, pp. 1–10.
[63] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proc. 7th Conf. Emerg. Netw. Exp. Technol.*, 2011, pp. 1–12.
[64] K.-F. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Adaptive weighted traffic splitting in programmable data planes," in *Proc. Symp. SDN Res.*, 2020, pp. 103–109.
[65] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," in *Proc. Conf. Commun. Archit. Protoc.*, 1991, pp. 133–147.
[66] Z.-L. Zhang, V. J. Ribeiro, S. Moon, and C. Diot, "Small-time scaling behaviors of internet backbone traffic: An empirical study," in *Proc. IEEE Conf. Comput. Commun.*, 2003, pp. 1826–1836.
[67] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker, "Contra: A programmable system for performance-aware routing," in *Proc. 17th USENIX Conf. Netw. Syst. Des. Implementation*, 2020, pp. 701–721.
[68] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "HULA: Scalable load balancing using programmable data planes," in *Proc. Symp. SDN Res.*, 2016, pp. 1–12.
[69] X. Li et al., "Detection and identification of network anomalies using sketch subspaces," in *Proc. 6th ACM SIGCOMM Conf. Internet Meas.*, 2006, pp. 147–152.
[70] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. Int. Conf. Database Theory*, Springer, 2005, pp. 398–412.
[71] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[72] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 323–336, 2002.
[73] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. Int. Colloq. Automata Lang. Program.*, Springer, 2002, pp. 693–703.
[74] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 109–118.
[75] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 561–575.
[76] L. Liu et al., "SF-sketch: A two-stage sketch for data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2263–2276, Oct. 2020.
[77] P4–16 language specification. [Online]. Available: https://p4.org/p4-spec/docs/P4--16-v1.2.1.html#sec-checksums
[78] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1094–1104, Oct. 2001.
[79] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.
[80] Y. Geng et al., "Exploiting a natural network effect for scalable, fine-grained clock synchronization," in *Proc. 15th USENIX Conf. Netw. Syst. Des. Implementation*, 2018, pp. 81–94.
[81] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise time-synchronization in the data-plane using programmable switching ASICs," in *Proc. ACM Symp. SDN Res.*, 2019, pp. 8–20.
[82] Y. Zhao et al., "LightGuardian: A full-visibility, lightweight, in-band telemetry system using sketchlets," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 991–1010.
[83] Y. Zhou et al., "Flow event telemetry on programmable data plane," in *Proc. Annu. Conf. ACM Special Int. Group Data Commun. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2020, pp. 76–89.
[84] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 63–74.
[85] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 221–235.
[86] The caida anonymized 2016 internet traces. [Online]. Available: http://www.caida.org/data/overview/
[87] Network Simulator (ns-3). [Online]. Available: https://www.nsnam.org/
[88] The NS.PY Discrete-Event Network Simulator. [Online]. Available: https://github.com/TL-System/ns.py
[89] Q. Yang et al., "DeepQueueNet: Towards scalable and generalized network performance estimation with packet-level visibility," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 441–457.
[90] A. Romanow, J. Mogul, T. Talpey, and S. Bailey, "Remote direct memory access (RDMA) over ip problem statement," *Request Comments (RFC)*, vol. 4297, 2005.
[91] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 537–549.