Concury: A Fast and Light-weight Software Cloud Load Balancer

Shouqian Shi
sshi27@ucsc.edu
University of California, Santa Cruz

Ye Yu*
Google

Minghao Xie
University of California, Santa Cruz

Xin Li*
Google

Xiaozhou Li
Celer Network

Ying Zhang
Facebook

Chen Qian
qian@ucsc.edu
University of California, Santa Cruz

ABSTRACT

A load balancer (LB) is a vital network function for cloud services to balance the load amongst resources. Stateful software LBs that run on commodity servers provide flexibility, cost-efficiency, and packet consistency. However, current designs have two main limitations: 1) states are stored as digests, which may cause packet inconsistency due to digest collisions; 2) the data plane needs to update for every new connection, and frequent updates hurt throughput and packet consistency. In this work, we present a new software stateful LB called Concury, which is the first solution to solve these problems. The key innovation of Concury is a new method to maintain large network states with frequent connection arrivals, which is succinct in memory cost, consistent under network changes, and incurs low update cost. The evaluation results show that the Concury algorithm provides 4x throughput and consumes less memory compared to other LB algorithms, while providing weighted load balancing and false-hit freedom, for both real and synthetic data center traffic. We implement Concury and evaluate it in two real networks. It achieves 67.2 Gbps single-thread throughput on a cheap desktop computer in 100GbE.

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; **Cloud computing**.

KEYWORDS

Load balancing; Software Load balancer; Cloud Computing; Data Center; Mobile Edge Computing

ACM Reference Format:
Shouqian Shi, Ye Yu, Minghao Xie, Xin Li, Xiaozhou Li, Ying Zhang, and Chen Qian. 2020. Concury: A Fast and Light-weight Software Cloud Load Balancer. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421279

1 INTRODUCTION

The load balancer (LB) is a fundamental network function of a data center that provides Internet services. To accommodate the high demand for popular service at scale, such as a search engine, email, photo sharing/storage, or message posting and interactions, a data center maintains multiple backend servers, each carrying a direct IP (DIP). For a particular service, clients send their requests to a publicly visible IP address, called the virtual IP (VIP). Each VIP is mapped to a pool of DIPs. An LB uses different DIPs to replace the VIP on the service requests and balances the load across the servers, so that no server gets overloaded to disrupt the service. An LB usually operates on or above layer 4.

Conventional hardware-based LBs [8, 10, 11] have limitations on scalability, availability, flexibility, and cost-efficiency [23]. Hence, major web services such as Google [23], Microsoft [32], and Facebook [7, 22] have started to rely on

*Ye Yu and Xin Li worked on this project during their PhD studies at University of Kentucky and UC Santa Cruz respectively.

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421279

stateful software LBs, which scale by using a distributed data plane that runs on commodity servers, providing high availability, flexibility, and cost-efficiency. A packet being *stateful* means that it belongs to a connection, and the prior packets of the connection have been forwarded to a DIP. Otherwise, the packet is *stateless*. The key functions of a stateful LB include the following. 1) For a stateless packet, which can be sent to an arbitrary DIP supporting its VIP, the LB algorithm should act as a *weighted randomizer* based on the current capacities of the backend servers. 2) For a stateful packet, the LB forwards it to the particular DIP that received the prior packets, preserving *packet consistency*.

The major challenge of a stateful LB algorithm is to preserve packet consistency under network dynamics, including new connection arrivals and DIP changes due to server failures or updates. Most existing LB algorithms use hash tables to store connection states as the data plane solution [23, 32]. These stateful LBs experience large memory cost of storing packet states or low capacity of packet processing. They require a large number of commodity servers to scale out, e.g., up to 3.5% - 10% of the data center size as reported by Microsoft [25] and Google [23]. Hence, some LBs use digests of connections (e.g., hash values of the 5-tuples of connection) rather than full connection states (such as 5-tuples) to reduce memory costs and improve throughput [23]. This design has two major weaknesses: 1) using long digests may still require a large amount of memory while using short digests causes violation of packet consistency due to digest collisions; 2) a massive number of new connections cause highly frequent data plane updates – a modern cluster may easily experience thousands of new connections per second [28] – which significantly hurts the packet processing throughput and possibly violates packet consistency. Existing methods relying on fast and concurrent reads and writes to hash tables [24, 26] cannot be easily applied to LB algorithms, because they only work with full state keys rather than digests. Recent work uses ASICs on programmable switches for fast table lookups [28], but further increases the infrastructure cost.

We propose the first stateful LB algorithm that resolves the current limitations, called Concury. [1] Its key innovation and contribution is a novel approach of *maintaining large-scale network states with a massive amount of newly arrived connections*, which is succinct in memory cost, consistent under network changes, and incurs extremely infrequent data plane updates. This approach could possibly be applied to many stateful network functions, such as NAT and LTE Evolved Packet Core, but this paper only focuses on LBs.

[1] The name Concury is from Concordia, the Roman goddess of balance and harmony, and Mercury, the Roman god of messages/communication and travelers, known for his great speed.

Compared to existing stateful LBs [7, 23, 25, 28, 32], Concury provides two main advantages. 1) We realize that the current limitations of software LBs are from the algorithmic designs for state maintenance and lookups: hash tables storing digests. To reduce memory cost, current LBs store the *digests* of states rather than the whole state identifier (e.g., > 100bits for a 5-tuple) [23, 28]. The drawbacks include 1) false table hits due to digest collisions [28], and 2) table explosion due to the difficulty of removing a digest. Concury uses a data structure to represent all packet states in a succinct manner (just two small arrays), by utilizing the theoretical foundation of minimal perfect hashing [18, 20, 21, 27, 39]. Concury is designed in such a way that it finds the specific destinations for stateful packets and *simultaneously* acts as a weighted randomizer for stateless packets with small memory cost and packet consistency. However, applying the theoretical Othello Hashing is *not straightforward*. There are several system building challenges to overcome: supporting efficient and fast lookup, managing connections under limited resources, and no false hits. 2) Data plane updates on existing LBs happen for every incoming connection. Concury includes the coordination between the data and control planes such that *Concury does not need to update its lookup tables for every incoming connection*. Instead, the Concury data plane is updated once every backend server change (DIP change), which happens much less frequently than new state arrivals. State maintenance and updates in Concury are much simpler than existing solutions, which allow Concury to maintain high lookup throughput and consistency.

In addition, Concury can be used for complex Internet applications and the emerging edge cloud [33, 34, 37, 38]. 1) It fits the condition of an edge cloud that typically has constrained resources – an edge LB may only be hosted by one server and could be co-located with other services on the server [34, 38, 41]. 2) Traditional cloud LBs consider a state for every TCP connection. In modern cloud or edge, states may be for *multi-connection* and at the device-level or process-level [34, 38, 41], i.e., the packets belonging to a single device should be sent to the same DIP. For example, a user device may keep offloading its video data to an edge server, let the server processes the data, and later request the analytical results from the server [34]. This whole process consists of multiple TCP flows and pseudo-flows with UDP, all of which should be sent to a consistent DIP. Unlike previous designs, the nature of Concury can easily support multi-connection states. 3) Modern cloud and edge servers might have heterogeneous capacities in computation, storage, and bandwidth [34, 38]. Concury reacts quickly to the weight changes due to failures or load dynamics of the servers.

We make several key intellectual contributions:

(1) The workflow of Concury is designed to achieve memory-efficiency, high throughput, load balancing, consistency, and false hit freedom.
(2) We propose a new method to maintain the dynamic set of states in the control plane and can instantly produce new lookup structures to update the data plane, under DIP pool changes.
(3) We add the functions of weighted randomizer and massive connection state maintenance to LBs.
(4) We implement Concury using DPDK [2] to shows its high performance **in two real networks**. We also build a P4 prototype to show its compatibility to programmable switches. The source code can be accessed here [14] and our results can be re-produced.

Concury achieves the **highest** packet processing throughput reported in literature (67.2 Gbps with single thread on a cheap desktop computer (<$800 not including the 100GbE NIC)) and low memory cost with 0 false hit, compared to existing stateful LBs. We consider Concury a major improvement because it achieves the best of three worlds: performance, cost-efficiency, and consistency (correctness). It fits new applications and systems. It also includes an ideal solution to dynamic state maintenance that is useful for other network functions.

The balance of this paper is organized as follows. Section 2 presents the related work. We formally model the LB algorithm in Section 3. Section 4 introduces some background algorithms. We present the detailed design of Concury in Section 5. The system implementation and evaluation results are shown in Section 6. We conclude this work in Section 7. This work does not raise any ethical issues.

2 RELATED WORK

An LB is an important component of a data center network, which distributes incoming traffic to different backend servers or other network functions [7, 23, 25, 32, 36]. Traditional hardware load balancers are expensive and not flexible. Hence, many large cloud services choose to use software load balancers [7, 12, 23, 25, 32]. In addition, LBs are also important for edge data centers [17, 34, 35, 41], which allow heterogeneous devices on the path to the remote cloud to offer storage and computing resources.

Stateful load balancers. Ananta [32] is a software stateful LB in a three-level architecture, which includes data center routers that run ECMP, a number of software multiplexers (SMuxes) on commodity servers, and a host agent on each backend server. However, each Ananta instance provides very slow packet processing speed as shown in [25]. Duet [25] makes use of forwarding and ECMP tables on commodity switches to store VIP-DIP mappings. Under frequent DIP pool changes, Duet may be not able to maintain PCC [28]. Maglev [23] is Google's distributed software load balancer running on commodity servers. The core algorithm of Maglev is to use a hash table to store connections as digests for load balancing and a new consistent hashing algorithm for resilience to DIP pool changes. SilkRoad [28] implements LB functions on state-of-the-art programmable switching ASICs, which requires more than 50MB SRAM. It supports high-volume traffic with low latency and preserves consistency. Deploying SilkRoad introduces extra hardware cost –each SilkRoad switch costs 6.5K USD, and multiple switches are needed for every cluster. In addition, both Maglev and SilkRoad may include false hits during connection lookups, due to the usage of digests rather than the complete state information. False hits cause two main problems. 1) A packet may be forwarded to a DIP that does not provide the correct service of its VIP and then fails. 2) Multiple states may share a digest in the table. It is difficult to decide when to delete a digest. Deleting the digest of a finished state might terminate an active state, if their digests collide. Hence the table size may explode over time, or some active states may be terminated. The typical data structure that can be used to maintain states in the above methods is Cuckoo Hashing [31]. Bonomi *et al.* proposed to use Approximate Concurrent State Machines (ACSMs) to maintain dynamic network states [19], but this method cannot be used for LBs. We compare Concury with existing stateful LBs in Table 1, where the experimental values are based on the DIP-V 16M-state network in 6.2.

Stateless load balancers. Beamer [30] and Faild [17] are recently proposed stateless LBs. Their forwarding logics do not store connection state but use a simple mapping algorithm (static or consistent hashing). They write a new field to every packet header to carry its DIP. The end servers need to examine *every* packet header to ensure that the packet is consistent with the state on this server. If not, the server performs overlay re-routing to the correct DIP. This method requires a kernel modification on the network stack of every server to add extra network processing. The computation and memory overheads are thus transferred to the server side and on a per-packet basis. Overlay re-routing might not be a significant problem when states are short-term. However, for multi-connection states that are long-term, stateless LBs may cause re-routing of most stateful packets, because after a duration the mapping would become very different. Compared to these methods, Concury only requires each server to run a lightweight state-tracking program in the application-layer, which does not change the network stack. Performance comparison of stateful and stateless LBs would be apple-to-orange because overhead occurs in different places. We do not intend to declare a clear victory between stateful and stateless LBs. The purpose of this work is to improve the

LB Algorithm	Lookup (Mpps)	Memory (MB)	Weighted LB	False hits	Packet type	Extra hardware	Update interrupt
ECMP + hash table (Ananta [32])	low	high	unclear	No	any type	No	frequent
Hash table w/ digest (Maglev [23])	14.63	18.63	Yes	exist	TCP only	No	frequent
Multi HTs w/ digest (SilkRoad [28])	16.11	4.36	No	exist	TCP only	ASIC	frequent
Concury (this work)	66.28	3.84	Yes	No	any type	No	infrequent

Table 1: Comparisons among stateful LB algorithms with example results. The numerical values are from the microbenchmark using 1M concurrent connections. More results can be found in § 6.

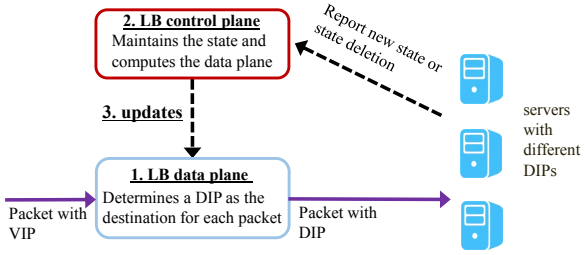

Figure 1: General model of a stateful LB

stateful LB design and leave the choice between stateful and stateless LBs to network operators.

3 SYSTEM MODELS AND OBJECTIVES

A service provided by a cloud/edge data center is identified by a publicly visible IP address, called virtual IP (VIP). The clients send their service requests to the VIP. An LB balances the load across the cloud/edge servers, so that no server gets overloaded and disrupts the service. Each backend server is identified by a direct IP (DIP). Hence, the core function of an LB is to map the VIP on a packet header to a DIP, based on the header information of the packet (e.g., its 5-tuple or other state identifiers). Each VIP is associated with its *DIP pool*, which includes the DIPs of the servers that provide the service identified by the VIP. The DIP pool of a VIP may vary depending on the service size and the environment (cloud or edge). If a server maintains the state of a packet, the packet must be sent to the DIP of the server. A state could be an ongoing connection or multi-connection.

Achieving all requirements of an LB stated in § 1 is challenging. Simple stateless algorithms (such as 'consistent' hashing) provide no guarantee of consistency. It is because the distribution algorithm needs to change when there is a DIP pool or weight change, and then stateful packets may be mapped to another server.

Recent stateful LB designs [23, 28] need to store connection state and ensure that all packets matching a state are consistently mapped to the same DIP. We summarize a general model of stateful LBs (as shown in Fig. 1), analyze the components of this model, and point out the design objectives.

1. LB data plane (LB-DP). The LB-DP processes packets and finds a DIP for each packet carrying a VIP. The DIP should be selected from the DIP pool behind the VIP, representing the set of servers providing the service of this VIP. The core algorithm should provide two functions: i) find the corresponding server (DIP) for each stateful packet, and ii) assign an available server (DIP) based on given weights for each stateless packet. The design objectives of the LB-DP is to achieve *high packet processing throughput* and *efficiency of memory cost*, because high-speed memory is a precious resource on both commodity servers (cache) and hardware switches (ASICs). In addition, the LB-DP should *balance the stateless packets based on the weights* reflecting the current capacity of each server, which may be heterogeneous and dynamic. For example, if a server is serving many large-size connections, it has to receive fewer new states than others in the near future. So we identify the 'weight' as an important input to the LB, and we expect that an LB acts as a weighted randomizer for new states.

2. LB control plane (LB-CP). The LB-CP receives the state changes from the servers, including new state establishments and state removals. Many existing designs use a TCP SYN packet as the indicator of a new state and allow LB-DP to notify the LB-CP directly [23, 28]. However, it does not work for UDP or multi-connection states. The design objectives of the LB-CP is to efficiently *maintain all state* of the incoming packets and quickly construct the new LB-DP to reflect *packet consistency* once an LB-DP update is needed. Ensuring that all packets of a connection are delivered to the same server is critical for LBs, because recovering a broken connection usually takes a long time and significantly hurts the user experience. In the edge or cloud where a unified data management layer is absent, packets from different flows of a single device should be sent to the same server. Achieving the device-level consistency could avoid overlay re-routing for many emerging applications such as media offloading.

3. Update. The LB-CP will notify LB-DP to make necessary changes under certain network dynamics, such as DIP pool and weight changes. The design objective of the update process is to *reduce the frequency of updating* because it will interrupt packet processing on the LB-DP.

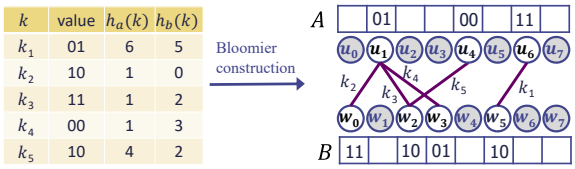

Figure 2: Construction of a Bloomier filter

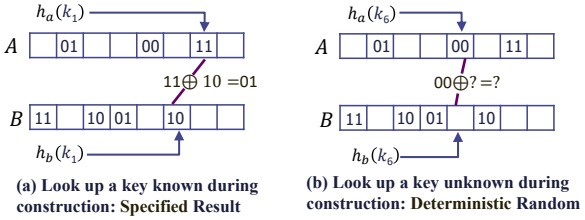

Figure 3: Lookups of Bloomier filter

Although Concury needs the servers to send state notification, the servers do *not* need to maintain any state just like prior stateful LBs. In other designs, server-to-LB messages are necessary for weighted load balance [25].

4 BACKGROUND ALGORITHMS

We propose to use the data structures and algorithms of minimal perfect hashing [18, 20, 21, 27, 39] for the Concury LB. One well-known perfect hashing based data structure is the Bloomier filters [20, 21]. The recently proposed Othello Hashing [39, 40] makes use of Bloomier filters to support forwarding information bases in programmable networks, including a variant of Bloomier filters as its data plane, a construction program in its control plane, as the interaction protocols of the two planes. Othello finds a setting of Bloomier filters to achieve good time/space trade-off for dynamic network environments. Though it was not designed for LBs, Othello qualifies as a great fit for LBs based on three reasons: 1) the lookup of Othello data plane is super fast and memory efficient; 2) the lookup is collision free, though no full key is stored in the data plane; 3) we designed an asynchronized update algorithm between the control plane to data plane, while keeping the PCC and weighted load balancing for all the time. We illustrate the first two points in this section, and the third poitn is detailed in § 5.

A Bloomier filter is not used as a filter, but a mapping for a set of key-value pairs. Let S be the set of keys and $n = |S|$. The lookup of each key returns an l-bit mapped to the key.

Bloomier filter construction in the Othello control plane. We use an example in Fig. 2 to show the Bloomier filter of a set of five key-value pairs. Each of the keys k_1 to k_5 has a corresponding l-bit value. Two arrays A and B are built with m_a and m_b elements respectively, where $m_a = n, m_b = 1.33n$. Each element of the arrays is an l-bit value. In this example, $l = 2$, and assume $m = m_a = m_b = 8$ for better illustration. For every value i in A we place a vertex u_i, and for every value j in B we place a vertex w_j. Two hash functions h_a and h_b are used to compute the integer hash values in $[0, m-1]$ for all keys. Then, for each key, we place an edge between the two vertices that correspond to its hash values. For example, $h_a(k_1) = 6$ and $h_b(k_1) = 5$, so an edge is placed to connect u_6 and w_5. For a key k and its corresponding value v, the requirement of Bloomier is that the two connected elements $A[h_a(k)] \oplus B[h_b(k)] = v$, where $\oplus$ is the bit-wise *exclusive or* (XOR). For key k_1 in this example, $u_6 \oplus w_5 = 01_2 = 1$. Vertexes colored gray represents "not care" elements. Note that after placing the edges for all keys, the bipartite graph, called graph G, needs to be *acyclic*. It is proved that if G is acyclic, it is trivial to find a valid element assignment such that the values of all keys are satisfied [39]. If a cycle is found, the construction needs to find another pair of hash functions to re-build G. It is proved that during the construction of n keys, the expected total number of re-hashings is < 1.51 when $n \leqslant 0.75m$ [39]. The expected time cost to construct G of n keys is $O(n)$, the time to delete or change a key is $O(1)$, and the time to add a key is amortized $O(1)$. The design can be trivially extended to $l > 2$.

Bloomier filter lookups in the Othello data plane. The Othello *lookup structure* is simply a Bloomier filter containing the two bitmaps A and B, as shown in Fig. 3 (a). To look up the value of k_1, we only need to compute h_a and h_b, which are mapped to position 6 of A and position 5 of B (starting from 0). Then we compute the bit-wise XOR of the two bits and get the value 01_2. Hence the lookup result is $\tau(k) = a[h_a(k)] \oplus b[h_b(k)]$.

The lookups are memory-efficient and fast. 1) The data plane only needs to maintain the two arrays. The keys themselves are not stored in the arrays. Hence the space cost is small ($2m/n$ per key). 2) Each lookup costs just two memory access operations to read one element from each of A and B. It fits the programmable network architecture: the data plane only needs to store the lookup structure, two arrays, while the control plane stores the key-value pairs and the acyclic bipartite graph G. When there is any change, the control plane updates the two arrays and lets the data plane to accept the new ones. When a Bloomier filter performs a lookup of a key that does not exist during construction, it returns an arbitrary value. For example in Fig. 3(b), $k_6 \notin S$ and its result may be an arbitrary value. We will utilize this property to construct a *weighted randomizer*.

It should be noted that updates may require re-hashing, which, although happens in low probability ($O(1/n)$), still takes $O(n)$ time and may introduce a notable latency to the control plane response time. Hence we propose an advanced

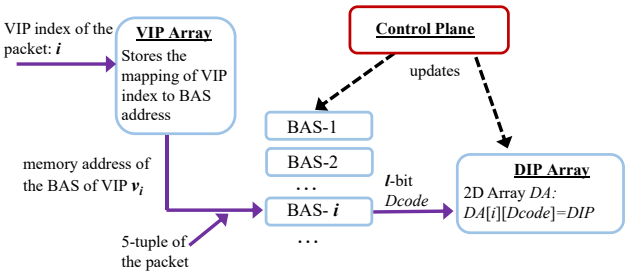

Figure 4: Workflow of Concury data plane

data structure called OthelloMap that always maintains an up-to-date lookup structure in the control plane to limit the response time to microsecond level, as explained in § 5.4.

5 DESIGN OF CONCURY

5.1 System overview

Notations. Let M be the number of VIPs in the network. Each VIP v_i is assigned an index i and its DIP pool contains t_i DIPs. The number of states of VIP v_i is n_i.

Concury follows the DP model introduced in § 3, including both the data plane and control plane. The input of the Concury data plane (Concury-DP) is a packet whose destination address is a VIP, and the output is the same packet whose destination has been replaced by a DIP. At each backend server (identified by a DIP), there is a lightweight application-layer program that tracks the current states at this server, which has been used for existing data center LBs [25]. The state tracking program will report the Concury control plane (Concury-CP) about new and terminated states. Concury-CP will update Concury-DP only when the DIP pool of a VIP changes, i.e., server failure/addition and server weight change. The update only applies to a small part of Concury-DP. The design objectives have been discussed in § 3.

Challenges of designing Concury. One key innovation of Concury is to abandon the conventional "lookup-then-distribute" workflow of prior LB designs and adopt a new approach that achieves 'lookup' and 'distribute' simultaneously. However, Bloomier and Othello were not originally designed for LBs. The challenges of applying Bloomier includes: 1) how to adjust Bloomier for both active state lookups and weighted randomizer; 2) how to design the data plane to minimize memory cost and maximize throughput; 3) how to resolve the false hits problem without modifying the server network stack; and 4) how to relax the requirement of updating for every new state in the data plane.

5.2 Concury data plane

Concury uses Bloomier filters as both a lookup structure to represent the state-to-DIP mapping and a weighted randomizer. As introduced in § 4, a Bloomier filter is built based on a set S of keys. In Concury, each key is the identifier of a state, i.e., 5-tuple. The value corresponding to a key is a DIP code (*Dcode*), which will be eventually converted to a DIP – the address of a backend server that holds the state. Note that a Bloomier filter provides the state-to-DIP mapping but does not actually store the keys. Hence the memory cost is significantly reduced.

There are two possible approaches to construct the state-to-DIP lookup structure of Concury. 1) All VIPs share a single lookup structure. 2) Each VIP has an individual Bloomier filter as the lookup structure, called a Bloomier array set (BAS), which stores only the state-to-DIP mapping of this particular VIP. This requires M BASes. We use this approach rather than a single and unified BAS because, 1) Upon change of a VIP's DIP pool, it is only necessary to update the Dcodes *this* VIP. The others are kept still. 2) Separating different VIPs further ensures a packet is not forwarded to a DIP in another VIP's pool. 3) Experimental results show that separate lookup structures provide 5% faster lookup speed than a unified one.

Note that maintaining per-VIP structures can also be used by other stateful LBs such as Maglev [23] to avoid the cross-VIP problem. However, it still cannot resolve the digest-deletion problem stated in § 2. Concury is unique because it can deal with both types of problems.

The workflow of Concury data plane is shown in Fig. 4, which includes three main steps. The lookup operation is *simple and fast*, including just four read operations and the hash computation.

Step 1. When Concury receives a packet, it first gets the VIP index i using the VIP v_i in the packet header, by either a table lookup or calculation. Since VIPs are determined by the edge/cloud operator, one can simply assign all VIPs with a single prefix, e.g., a 22-bit prefix, then the last 10 bits of a VIP can be used as the VIP index, supporting 1K VIPs. Concury maintains a *VIP array* that stores the memory addresses of different BASes, using a static array whose index is the VIP index. The result of Step 1 is the memory address of the BAS of VIP v_i. The array is small and static.

Step 2. Using the memory address from Step 1, Concury finds the BAS for VIP v_i, denoted as BAS-i. BAS-i only includes the two arrays A and B to support the calculation of the lookup result $\tau(k) = A[h_a(t)] \oplus B[h_b(t)]$, where t is the 4-tuple of k, without the destination IP address compared to the 5-tuple. The result is an l-bit value called DIP code, denoted as *Dcode*. Each DIP code will be mapped to an actual DIP in Step 3, and it is a many-to-one mapping. Two different DIP codes may be mapped to a single DIP.

Step 3. This step finds the actual DIP using the l-bit *Dcode*. Concury maintains a 2D array called DIP array, denoted by *DA*. The element $DA[i][Dcode]$ is the DIP of the *Dcode* for VIP v_i. This 2D array is independent of the number of current states and does not cost much memory. Assume there are

512 VIPs and $l = 12$. The memory cost is about 2MB. Note $DA[i][Dcode]$ for any l-bit value of $Dcode$ is a valid DIP of the VIP v_i. To further reduce the memory cost, $DA[i][Dcode]$ can be a DIP index that can be transferred to an DIP with one more static table lookup.

Data plane complexity analysis and comparison.

1) Time cost. Concury-DP is very simple and fast. Each lookup is in $O(1)$, including *at most* 6 read operations from static arrays, 2 hash computations (32 bits for each), and an XOR computation. This cost is smaller than Cuckoo+digest, a commonly used LB table design [23, 28], which needs more read operations and hash computations for both stateful and stateless packets.

2) Space cost. Let n be the number of total states, l_d be the length of Dcode, and l_v be the length of the DIP index in the DIP table. The total memory cost of Concury-DP is $2.33l_dn + 64m + 2^{l_d}l_vm + 48 \cdot 2^{l_v}$ bits, which is much smaller than that of Cuckoo+digest in practical setups.

5.3 Weighted load balancing

Reason for using DIP code. One may notice that to process the first packet of a new state, Concury gets $Dcode$ and then translates it to the DIP, rather than directly putting the DIPs as the lookup results of a BAS. Our method reduces the storage cost because a DIP is 32-bit long, while a DIP code can be much shorter, e.g, 10 bits. The total number of distinct DIP codes, 2^{l_d}, can be larger than the number of DIPs, e.g., by more than an order of magnitude, in order to provide the granularity for a weighted randomizer. The $Dcode$ to DIP mapping is determined by how the LB wants to assign the weights among DIPs of this VIP. For example, if $Dcode$ has 4 bits and there are 4 DIPs and all DIPs have equal weights, then we may map $Dcode$ in [0000, 0011] to DIP_1, $Dcode$ in [0100, 0111] to DIP_2, $Dcode$ in [1000, 1011] to DIP_3, and $Dcode$ in [1100, 1111] to DIP_4. We may consider $Dcode$ as a ball and each DIP as a bin.

How to achieve weighted load balancing. We first show that for an unknown state, the probability that a BAS will return a particular $Dcode$ is uniformly distributed among all possible values of $Dcode$.

For a new state c, the lookup result of a BAS is $Dcode = \tau(c) = A[h_a(c)] \oplus B[h_b(c)]$, where $A[h_a(c)]$ and $B[h_b(c)]$ are both l-bit values. Assume that $A[h_a(c)]$ ($B[h_b(c)]$) has equal probability to be any element in array A (array B), which is true if h_a and h_b are uniform hashes. Each element in A or B can be either 'determined' or 'free'. A determined element corresponds to a white vertex as in the example of Fig. 2, whose value should be fixed during the construction to provide correct lookups for current states. A 'free' element corresponds to a gray vertex and its value is 'not care'. We assign uniformly random values for every free element. As a result, if $A[h_a(c)]$ and $B[h_b(c)]$ are both determined, $Dcode$ is determined. If one of $A[h_a(c)]$ and $B[h_b(c)]$ is free, then $Dcode$ is random. We know that A and B both have m elements and there are m^2 possible pairs of $A[h_a(c)]$ and $B[h_b(c)]$. Among them, only n pairs produce determined values of $Dcode$ and the portion is $n/m^2 < 1/n$. Hence, only a small portion of the results are determined, and the others can be considered uniformly random.

We use empirical results to validate this uniformity. Fig. 5 shows one typical example. We let the value length $l = 10$. Hence, there are 1024 possible Dcodes. We enumerate all possible combinations of indexes of A and B and compute the resulting Dcodes. The hash functions used in Concury is CRC32. We observe that using Concury, the combinations (stateless packets) are very evenly distributed to different Dcodes, with min, 10%, mean, 90%, and max values to be 925, 980, 1024, 1066, and 1120 respectively. Results of other experiments are similar.

We compare Concury with MD5 and SHA256. Although MD5 and SHA256 are not strictly uniform, they are considered *sufficiently uniform* in practice. We show that Concury is comparable to them in uniformity and is sufficiently good to use in practical systems. We conduct two well-known statistical tests, the chi-squared test and Kolmogorov-Smirnov test, to compare Concury, MD5, and SHA256 with the uniform distribution. As shown in Fig. 6 and 7, each of them fails around or less than 10% of the tests, because they are not strictly uniform. Concury is no worse than either MD5 or SHA256, especially when $l_d > 11$ (Dcode count > 2048). In our implementation, we set $l_d = 12$. We will further evaluate the load distribution to DIPs in § 6.4.

Based on the uniform Dcode distribution, we may use the Dcode-DIP mapping to implement a weighted randomizer. The number of Dcode should be larger than the number of DIPs by a certain scale, e.g. >8x. Then, the weight of a DIP is reflected by the number of entries in the table DA. For example, if DIP d_1 has weight 1.0, and DA holds 100 entries pointing to d_1, then for DIP d_2 with weight 2.0, DA should hold 200 entries pointing to d_2. If d_2 is near full, which is unlikely, then the weight of d_2 should be lowered to reflect its current remaining capacity, and new connections go to d_2 with a smaller probability. This weight change will incur a full synchronization between the control plane and the data plane of Concury, which is detailed in § 5.5.

5.4 Concury control plane

The tasks of the Concury control plane (Concury-CP) are two-fold: 1) tracking existing states; and 2) generating new data plane structures, mainly the new BASes, when a data plane update is required. A naïve solution is to use a hash table to store a set of state-DIP pairs. When an update is

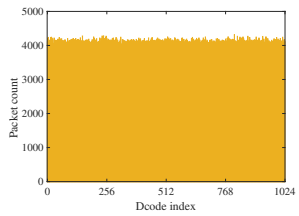

Figure 5: Stateless packet distribution by Dcode

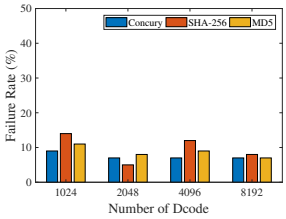

Figure 6: Chi-squared test

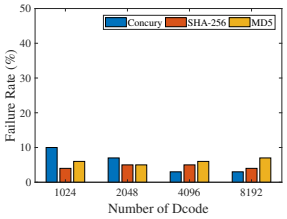

Figure 7: Kolmogorov-Smirnov test

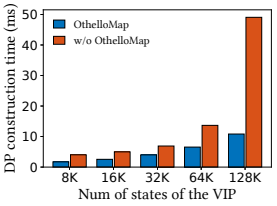

Figure 8: Response time of Concury-DP construction

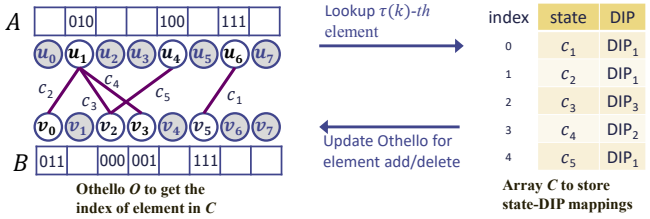

Figure 9: Lookup/update of an OthelloMap

needed, the new BAS is constructed from the set. Our *innovative idea* is to design a new data structure called the OthelloMap that maintains both the state-DIP pairs and the BASes for all current states. Note if the network includes M VIPs, the control plane has M OthelloMaps. The purpose of using OthelloMap is to quickly generate a new Concury-DP when a network dynamic happens.

Components of an OthelloMap. As shown in Fig. 9, an OthelloMap of VIP v includes two parts. 1) An array C of size n, where n is the number of current states of VIP v. Each element of C stores a state-DIP pair. 2) A BAS O constructed using the set of current states. The lookup result of O, using the state identifier (ID) c, is the index i such that $C[i]$ stores the state-DIP pair of c. Note the length of i is no smaller than $\lceil \log_2 n \rceil$ bits.

Set query to OthelloMap. The set query is a basic function of OthelloMap. The input is a possible state ID c', and the output is either the corresponding DIP or 'not exist'. To conduct a set query, the OthelloMap performs a lookup to the BAS O using c' and get a value i. If the state exists, $C[i]$ includes the DIP. Otherwise, the connection stored in $C[i]$ does not match c'. Hence, it can return 'not exist'. This process takes $O(1)$ time.

Addition/deletion to OthelloMap. To add a state-DIP pair $\langle c, DIP \rangle$, to the OthelloMap, we first apply a set query of c. If c exists, $C[i]$ is revised to $\langle c, DIP \rangle$. If c does not exist, we store $\langle c, DIP \rangle$ to $C[n+1]$. Then we add $\langle c, n+1 \rangle$ to BAS O. This process takes amortized $O(1)$ time. To delete a state-DIP pair $\langle c, DIP \rangle$ from the OthelloMap, we apply a set query of c. If c does not exist, we do nothing. Otherwise, c and its DIP are stored in $C[j]$. We delete them from $C[j]$ and move the element in $C[n]$, say $\langle c', DIP' \rangle$, to $C[j]$. Then we revise the value corresponding to c' in BAS O from n to j. This process takes $O(1)$ time.

Memory cost analysis of Concury-CP. Let l_i be the length of the index i and l_k be the length of each state-DIP pair information. The memory cost of Concury-CP is $2.33 l_i n + (l_k + l_d)n + 64m + 2^{l_d} l_v m + 48 \cdot 2^{l_v}$, where $2.33 l_i n$ is the overhead of the BAS O, $(l_k + l_d)n$ is the overhead of the array C, and the remaining is for the VIP array and DIP array that need to be updated to the data plane.

Performance gain using OthelloMap. We compare the time to construct a new DP with and without OthelloMap. The results are shown in Fig. 8. *OthelloMap significantly reduces the response time in the control plane during Concury updates* by over 50%.

Interaction of Concury-CP and Host Agents. Concury-CP receives state arrival/termination reports from Host Agents running on different DIP servers. Upon receiving a report, Concury-CP performs corresponding addition/deletion operations to the corresponding OthelloMap.

We discuss Task 2 of Concury-CP, i.e., how Concury-CP generates new data plane structures for network updates in the next subsection.

5.5 Reactive control/data plane update

Concury-CP does not have to update the Concury-DP on receiving state arrival/termination reports. Instead, it only updates the Concury-DP when there is a DIP-pool change. It is because only under a DIP-pool change, the current Concury-DP may violate consistency. Recall that Concury-DP includes the VIP array, the BASes for all VIPs, and the DIP array. For the change on a DIP pool of VIP v_i, only the BAS related to v_i and the i-th dimension of the DIP array needs to be updated, which are a relatively small portion of the entire Concury-DP. All other parts can be kept still.

Updating the DIP array is based on the load balancing method introduced in § 5.3, which is fast. To generate the updated BAS of v_i, denoted by O_i, we need to include all current states and remove terminated ones. The BAS of the OthelloMap of v_i, denoted by O_i', includes all states. The only

difference between O_i and O'_i is their lookup values (*Dcode* versus OthelloMap index). Recall that the main computation complexity of BAS construction is to compute the acyclic bipartite graph G to include the set of keys. Once G is determined, assigning the values of the keys can be done by starting from either end of the component, with complexity bounded by a one-time pass of the values. Therefore we simply re-use the G from the OthelloMap and assign the *Dcode* values, which takes a short and bounded time. In the end, Concury-CP sends the updated structures to Concury-DP using a programmable network API.

Upon receiving the update message, Concury-DP only needs to modify the arrays related to one particular VIP. Since the memory spaces of all VIPs are independent, the modified memory size is very small (less than 1MB in most cases). The packets to other VIPs can be concurrently processed while updating the data plane. In addition, we design the *concurrent control* method that locks 1024 bits at the same time for updating and only blocks packet lookups that need to access the 1024 bits. Due to space limitations, we skip the details.

Update complexity. The time/space complexity of data plane update is in $O(l_d n_i)$, where n_i is the number of connections of VIP v_i and l_d is the length of Dcode. Note Concury updates happen infrequently, once per DIP change, and only apply to the part of data plane structures of one VIP.

5.6 Consistency guarantee under dynamics

An LB experiences three types of dynamics: 1) state arrival/termination; 2) DIP pool changes; 3) VIP changes. It is important that packet consistency is still preserved during network dynamics. For state arrival and termination, Concury-DP has no change. In this case every packet to a VIP i will have three possibilities for the BAS lookup.

1) The state ID of the packet, k, is known by Concury-CP during the construction of BAS-i, and the value of looking up k is *Dcode* which can be mapped to the DIP holding this state. Then the lookup result $\tau(k) = Dcode$ and the packet will be forwarded to the correct DIP.

2) The state ID k is unknown by Concury-CP during the construction, and the packet is the first one of a new state. Then according to the property of BAS, $\tau(k)$ is an arbitrary l-bit *Dcode*. According to the property of the table *DA*, $DA[i][Dcode]$ always stores a valid DIP for VIP v_i. Hence the packet will be forwarded to a valid DIP D.

3) The state ID k is unknown by Concury-CP during the construction, and the packet is not the first one of a new state. Hence the first packet was processed after the latest construction and update, which was forwarded to a DIP D. Since the data plane has not been updated since then, Concury still returns D as the DIP of this packet, which preserves consistency.

Concury does not cause false hits either. Using the three-level lookup structure, for any new TCP packet or UDP packet, the corresponding BAS will return a *Dcode* that will be mapped to a valid DIP.

When a DIP pool change happens, the *Dcode* to DIP mapping needs to be adjusted. Again using the example in Section 5.2, we may map *Dcode* in [0000, 0011] to DIP_1, *Dcode* in [0100, 0111] to DIP_2, *Dcode* in [1000, 1011] to DIP_3, and *Dcode* in [1100, 1111] to DIP_4. The state c is mapped to 0100 and hosted on DIP_2. Suppose DIP_4 fails, and the mapping is adjusted as: *Dcode* in [0000, 0100] to DIP_1, *Dcode* in [0101, 1001] to DIP_2, *Dcode* in [1010, 1111] to DIP_3. Then the corresponding values in *DA* should be adjusted, e.g., $DA[i][0100]$ should be changed to DIP_1 from DIP_2. Also, packets of the state c should stick to DIP_2, and hence we change its *Dcode* to 0101 and revised the BAS accordingly. In this way, packet consistency is preserved.

VIP changes are very infrequent and can be handled easily. It requires only adding an element to the VIP array and adding/deleting corresponding BAS and one dimension of the DIP array. No packet consistency is involved.

Concury achieves packet consistency without requiring updating for every new state. It only updates when there is DIP change. This is a unique feature of Concury compared to other stateful LBs to achieve processing and update efficiency.

There is a possible consistency violation when a packet of a new state arrives during a Concury update. This is a common problem for all software LB designs. We let Concury buffer all stateless packets during updates. Note compared to other methods that update on a per-connection basis, Concury updates on a per-DIP-change basis, hence such a problem happens very infrequently.

6 IMPLEMENTATION AND EVALUATION

6.1 Evaluation methodology

We conduct three types of evaluations: 1) algorithm micro-benchmark; 2) Concury prototype using DPDK [2] deployed in two real networks (100GbE lab network and CloudLab [1]), and 3) a P4 prototype running on Mininet [16]. **Our code is publicly available with an anonymous link [14]. The results can be reproduced.** The purpose of the algorithm micro-benchmark is to compare the algorithms of Concury over existing solutions thoroughly. The purpose of evaluating software LB with DPDK is to show the actual performance of Concury running in real networks. The purpose of the P4 evaluation is to show that Concury can also be deployed to programmable switches.

We compare Concury with two recent stateful LB algorithms: 1) Hash table with digest, used in Maglev [23]; and

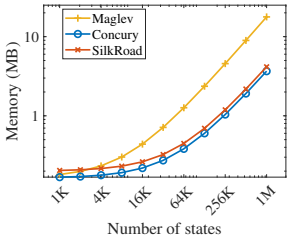

Figure 10: Memory cost for DIP-E and Small network

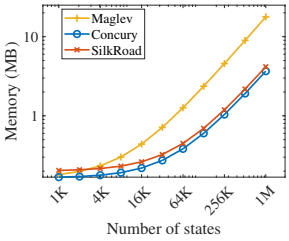

Figure 11: Memory cost for DIP-V and Small network

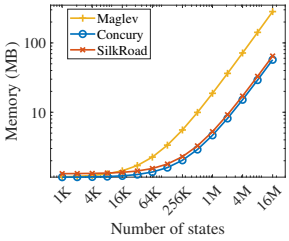

Figure 12: Memory cost for DIP-E and Large network

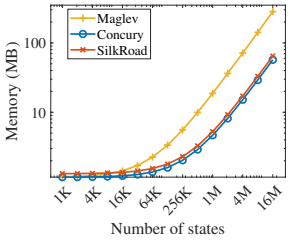

Figure 13: Memory cost for DIP-V and Large network

2) Multi hash tables with digest, used in SilkRoad [28]. Note SilkRoad was designed for special hardware, i.e., programmable switch ASICs with > 50MB memory. Hence, the performance shown in [28] is different. Since Maglev and SilkRoad are not open-source, we implement their LB algorithms *in our best effort to improve their performance and ensure consistency*, but we are not able to re-build identical system prototypes of Maglev and SilkRoad as some of their technique details are not fully presented [23, 28]. In addition, we also separate the hash table of Maglev in a per-VIP basis –a fix to reduce potential digest collisions but not fully resolving it. We evaluate the performance metrics, including memory cost, processing throughput, and load balancing. For all experiments, we verify that packets of a single state are always sent to a single DIP. Concury causes neither packet consistency violation nor false hits, hence we do not spend space to show them further. We do not compare Concury with stateless LBs [17, 30], such as Beamer [30]. It is because the main overhead of stateful and stateless LBs are at different places: network function side vs. server side. Also, stateless LBs require to change the server stack, whose cost is difficult to measure. Hence it is hard to conduct a toe-to-toe comparison.

We use CRC32-C [6] for robust and faster hash results in Concury. Recall that the construction of BASes may need sufficient different hash functions. We generate these hash functions using the following approach. Let H be a CRC32 hashing and seed be a 32-bit integer. We let $h_a(k) = H(k, \mathsf{seed}_a)$ and $h_b(k) = H(k, \mathsf{seed}_b)$. Thus, h_a and h_b are uniquely determined by seed_a and seed_b, respectively.

We use the **real traffic trace** from the Facebook data center networks [9] for experiments. Since the packets in the trace only carry the DIPs, we assign them to 128 VIPs. We also generate synthetic traffic for production runs and dynamic experiments over a duration of time. We generate two settings of the synthetic traffic: 1) *DIP-E*. All VIPs have the same number of DIPs, and they have the same number of concurrent states at any time. 2) *DIP-V*. VIPs have varied numbers of DIPs, and the numbers of concurrent states also vary with the numbers of DIPs. The number of VIPs may be 128 or 256. We also consider two types of networks: The *Small* network models an edge, and the *Large* network models a cloud. In the Small network, each VIP has 32 DIPs for DIP-E and 8 to 64 DIPs for DIP-V (32 on average). In the Large network, each VIP has 128 DIPs for DIP-E and 32 to 256 DIPs for DIP-V (128 on average). We vary the number of states from 1K to 16M for Large and 1K to 1M for Small, which covers the range of practical networks. According to actual measurement [28], the 99th percentile number of concurrent connections in the PoP cluster of a large web service provider is smaller than 10M. Other types of clusters and edge networks have fewer active states, varying from a few thousands to 10M.

For most experiments, we conduct production runs for at least 20 times and take the average. The variations are little and difficult to show in the figures.

6.2 LB algorithm evaluation

Algorithm implementation details. We have implemented the complete functions of both Concury-DP and Concury-CP on a commodity desktop server with Intel i7-6700 CPU, 3.4GHZ, 8 MB L3 Cache shared by 8 logical cores, and 16 GB memory (2133MHz DDR4). Different components of Concury interact as in Fig. 4. In addition, we need to provide a series of packets from different states and let Concury process them. One straightforward approach is to feed the LB with an existing traffic trace. However, the time for transmitting the data from the physical memory to the cache is too long compared to the packet processing time on Concury. Hence, we use a linear feedback shift register (LFSR) to generate the states (identified by the 5-tuple) of every packet. The generated states are uniformly distributed over all possible 5-tuples, which is the worst case for load balancing performance for the lack of time locality. One LFSR generates about 200M states (5-tuples) per second on our server. In addition, we provide event-based simulation using real traffic data to study the processing delay on Concury. Note that LFSR gives no favor to Concury because the states are generated in a round-robin scenario, which provides the minimum cache hit ratio. We use 1883 lines of C++ code in total for this prototype.

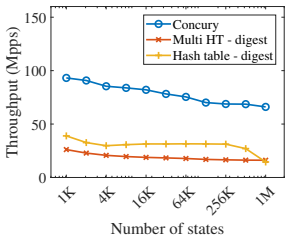

Figure 14: Throughput for DIP-E and Small network

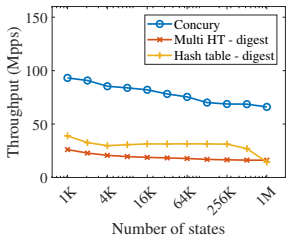

Figure 15: Throughput for DIP-V and Small network

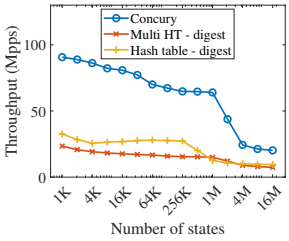

Figure 16: Throughput for DIP-E and Large network

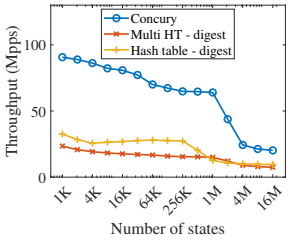

Figure 17: Throughput for DIP-V and Large network

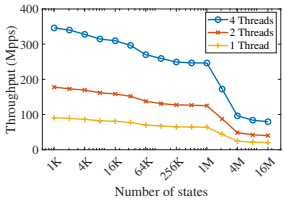

Figure 18: Throughput for multi-thread

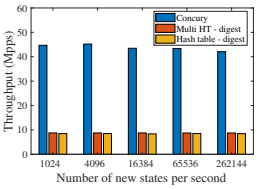

Figure 19: Throughput during data plane updates

Memory efficiency. Fig. 10 and 11 show the memory cost of the LB algorithms of Concury, Maglev, and SilkRoad in Small networks for the DIP-E and DIP-V setups respectively. The memory cost of Concury is less than 1MB for <256K states and 4MB for 1M states. The memory is only 20%-30% of that of Maglev, when the number of states is >64K. It is very close to that of SilkRoad. We also show the memory cost results in Large networks in Fig. 12 and 13. Concury has similar advantages compared to Maglev. When there are 8M concurrent states, both Concury and SilkRoad use < 38MB. The memory cost for the DIP-E and DIP-V setups are similar. Concury is very efficient in terms of memory cost: it can be implemented on hardware switches with limited programmable ASICs or commodity servers that have limited caches. Both Maglev and SilkRoad use digests, which introduce false hits. Concury provides false-hit freedom using similar or less memory.

Processing throughput. The processing throughput of an LB algorithm characterizes its capacity. With higher throughput, the network needs to deploy fewer instances of the LB, and the infrastructure cost is reduced. Fig. 14 and 15 show the throughput of the LB algorithms of Concury, Maglev, and SilkRoad in Small networks, using *a single thread* on a commodity desktop, for the DIP-E and DIP-V setups respectively. The metric is in millions of packets per second (Mpps). Note SilkRoad was designed for programmable switch ASICs. We implement the algorithm used in SilkRoad, named 'Multi-level Hash Tables with Digest' (Multi HT-digest), on commodity servers, and compared it to Concury. Similarly, we also implement the algorithm used in Maglev, named 'Hash Table with Digest'. Concury achieves > 65Mpps when the number of concurrent states is < $1M$ and shows > 2x advantage compared to Hash Table with Digest and Multi HT-digest. For Large network results shown in Fig. 16 and 17, when the number of states is > $1M$, the throughput reduces because the memory size is larger than the CPU cache size. However, Concury still maintains the > 2x advantage in throughput. The main reason resulting in the throughput advantage of Concury is that the data plane of Concury requires simpler operations than others. In addition, Fig. 18 shows the throughput of Concury scales well with the number of threads: it reaches > 250Mpps with < $1M$ states. The threads share the same memory space and do not compete for cache space. To validate that the performance is not CPU-dependent, we perform the same experiments on a workstation with Intel Xeon CPU E2-2687W. In all experiments, Concury shows higher throughput than others. The results are not shown due to space limitations.

Cost of data plane update. Data plane updates consume CPU time. Hence, on a single thread, if data plane updates are complex, the throughput will evidently downgrade. Existing LBs have no concurrent read/write designs [23, 28]. We conduct the following set of experiments to evaluate the impact of updates to Concury-DIP performance. We set the number of concurrent states to 1M and let new states join the network. The arrival rate ranges from 1K per second to 256K per second, reflecting the arrival rate in real networks. The DIP pools also change once per 10 seconds. The throughput during updates is shown in Fig. 19. The throughput of Hash table-digest (in Maglev) and Multi HT-digest (in Silkroad) clearly downgrade (to <10Mpps) compared to the results shown in the static experiments in Fig. 17. Concury experiences downgrading too (to 42Mpps), but the impact is limited. Hence, the data plane update cost of Concury is small compared to other methods.

Response time and scalability of Control plane update. We show the performance of Concury-CP in two aspects: 1) Response time of a DIP/weight change; and 2) Update time for new states. When a DIP/weight change happens, both the control and data planes need to be updated to reflect

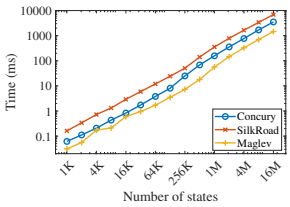

Figure 20: Time to insert new states to control plane

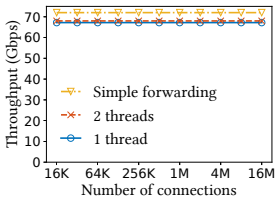

Figure 21: Throughput on DPDK

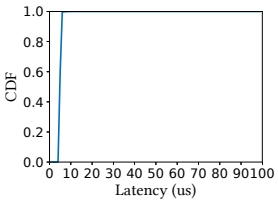

Figure 22: CDF of processing latency on DPDK

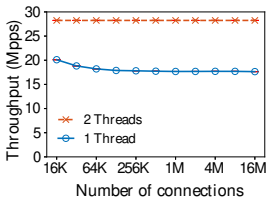

Figure 23: Throughput on DPDK in CloudLab

the change. Concury-DP provides a tremendous advantage in response time by leveraging OthelloMap as shown in Fig. 8. We find that when there are 8K to 128K states for one VIP (1M to 16M in total), the Concury-CP response time is only 2-12ms. On the other hand, Maglev requires very complex updates because it uses digests rather than the entire keys in the hash table. We further show the time cost of inserting new states to the control plane in Fig. 20. Note both the x and y axes are in logarithmic scale, and all three curves increase linearly with the number of the new states. For 16M new states, it only takes Concury a few seconds to complete all updates. Hence, Concury-DP is sufficiently fast and scalable to complete updates.

6.3 Evaluation of Concury in real networks

Implementation details. We implement Concury as a software LB using Intel Data Plane Development Kit (DPDK) [2] in two real networks: 1) a lab 100GbE built by Mellanox MCX516A-CDAT NICs and 2) CloudLab [1]. DPDK is a series of libraries for fast user-space packet processing [2]. DPDK is useful for bypassing the complex networking stack in Linux kernel and it has utility functions for huge-page memory allocation and lockless FIFO, etc. We modified the code of Concury-DP and link it with DPDK libraries.

6.3.1 100GbE in the lab. To build a lab 100GbE, we connect two commodity servers (called Node 1 and Node 2) back-to-back to construct the evaluation platform of Concury. Each of the two nodes is equipped with a Dual-port Mellanox MCX516A-CDAT NIC, which provides 2x100Gbps duplex bandwidth. There are 16 lanes of PCIe V3.0, which only support a bandwidth of duplex 120Gbps between the NIC and the CPU. Each node has an Intel i7-6700 8-core CPU at 3.40GHz and costs <$800 and each NIC costs $800. The Ethernet connection is 2x100Gbps.

Logically, Node 1 works as both a series of clients and a number of backend servers (DIPs) in the cloud, and Node 2 works as the software LB. Node 1 uses the DPDK official packet generator Pktgen-DPDK [5] to generate random packets and sends them to Node 2. The 5 tuple of the generated packets are uniformly randomly distributed, which exhibits the least locality in memory access and shows the lower bound performance of Concury. Concury is deployed on Node 2 and forwards each packet back to Node 1 after determining and rewriting the DIP of the packet. Node 1 then checks the packet consistency to DIPs and records the receiving bandwidth as the throughput of the whole system.

In the real network, the results show that the Concury software LB achieves 100% packet consistency and the load balancing results are identical to those in § 6.2.

Fig. 21 shows the throughput of Concury for DIP-V traffic, measured in Gbps, where every packet is 256 bytes long, same to the experiments of Maglev [23]. We first evaluate the maximum capacity of the platform by a simple forwarder that reads the 5-tuple of each packet and transmits it to the incoming port without looking up any FIB or table. The *maximum capacity* is 72.02 Gbps.[2] We evaluate up to 16M concurrent connections in the LB as shown in Fig. 21. On a single thread, Concury can process 67.20 Gbps (93% of the maximum capacity). **We do not find a better single-thread software LB throughput in the literature.** Using 2 threads, Concury improves little towards the maximum capacity, and the bottleneck is thus not on the Concury LB algorithm. We expect a much higher throughput of multi-thread Concury if it is deployed on servers with more powerful NICs and memory buses. Fig. 22 shows the CDF of the algorithm processing latency of Concury. The latency is on a 24-packet batch basis. We collect the latency information by recording the time before fetching a batch of the packets and after sending out all packets in the batch. > 99% batches finish less than 7 us.

6.3.2 CloudLab. CloudLab [1] is a research infrastructure to host cloud computing experiments. Different kinds of commodity servers are available from its 7 clusters. We use two nodes c220g2-011307 (Node 1) and c220g2-011311 (Node 2) in CloudLab to construct the evaluation platform of Concury software LB prototype. Each of the two nodes is equipped with one Dual-port Intel X520 2x10Gbps NIC, with 8 lanes of

[2] 72.02 Gbps equals to 35.16 Mpps. We find it is common that the maximum transmission capacity is less than the NIC bandwidth. For example, Maglev [23] deployed by Google shows that its maximum capacity on a 10GbE NIC is 12 Mpps (=6.14 Gbps).

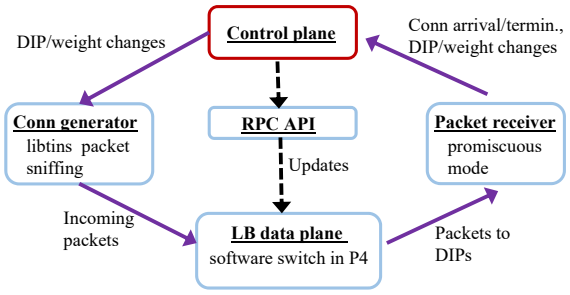

Figure 24: P4 prototype on Mininet

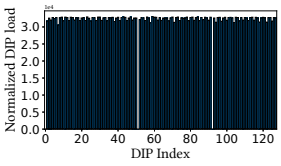

Figure 25: Normalized DIP load by P4 (real traffic)

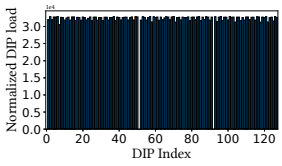

Figure 26: Normalized DIP load by P4 (synthetic traffic)

PCIe V3.0 connections between the CPU and the NIC. The switches between the two nodes support OpenFlow [29] and are claimed to provide full bandwidth. Fig. 23 show that the Concury in CloudLab also achieves 100% packet consistency. On a single thread, Concury can process and forward at least 17.63 Mpps (62.5% of the maximum capacity). Using 2 threads, Concury can achieve the maximum network capacity of the node. As a comparison, hash table based method cannot achieve the network capacity by 2 threads.

6.4 Evaluation on P4 prototype

We also build a P4 prototype of Concury, in which the data plane includes around 400 lines of P4 code. The prototype is based on the simple switch behavioral model [3] of the $P4_{16}$ language [4]. To manage data plane tables, we add a middle layer between the data plane and control plane with C++ Thrift remote procedure call (RPC) API provided by library PI [13].

We use Mininet [16] to implement the experimental platform to run Concury, which includes a P4 switch as the Concury LB, a Concury control plane program, a host to generate packets from clients, and a host representing 16K logical DIPs, as shown in Fig. 24. The receiving host uses the promiscuous mode to accept packets with different DIPs. We use libtins network packet sniffing library [15] to generate and send packets. To allow the control plane to communicate with the data plane through RPC, we add the NAT support to the prototype; hence the host can access TCP ports of the physical machine. We use the P4 prototype to evaluate the load balancing of Concury using both real and synthetic traffic. Given that Concury shows significant improvement over SilkRoad on software LB as shown in § 6.2 and Concury-DP is no more complex than that of SilkRoad, we expect Concury's throughput on a hardware switch may be no worse than that of SilkRoad. The results of load balancing should be consistent on both Mininet and hardware switches.

In this set of experiments, every VIP has 128 DIPs and DIPs have different weights, which reflect their resource capacities, to receive new connections. We use each connection to represent a state. We define a metric L, called the normalized DIP load, as $L = c_i/w_i$ where c_i is the number of connections forwarded to DIP_i and w_i is the weight of DIP_i. We show the normalized DIP load inside one VIP in Fig. 25 and 26 for real and synthetic traffic respectively. We find that the loads for DIPs are evenly distributed. Two DIPs showing 0 are with weight 0. The results of the other VIPs are very similar.

6.5 Summary of evaluation

As stated in § 3, the design objectives include the high packet processing throughput, efficiency of memory cost, weighted load balancing, quick construction, and packet consistency. Concury performs well in all aspects. Compared to prior solutions, Concury shows the advantages in all these aspects and is only weaker in inserting new states as shown in Fig. 20. The insertion speed is still sufficiently good for large cloud networks. In addition, Concury is a portable solution and does not rely on any specific platform.

7 CONCLUSION

We design and implement a new software stateful LB called Concury, which achieves weighted balancing of incoming traffic, maintaining consistency, high throughput, memory efficiency, and false hit freedom. It satisfies the requirements of a load balancer for cloud and edge data centers. Concury represents connection states without storing the actual state information and incurs low update cost. We implement Concury on both software and P4 prototypes and evaluate it in two real networks. Evaluation results show that Concury provides higher packet processing throughput by >2x and lower memory cost compared to existing stateful LB algorithms. In real network experiments, Concury achieves the highest packet processing throughput reported in literature. Our future work will be extending Concury to mobile client environments.

8 ACKNOWLEDGEMENT

S. Shi, Y. Yu, X. Li, and C. Qian were partially supported by NSF Grants 1701681, 1717948, and 1932447.

REFERENCES

[1] [n.d.]. CloudLab. https://www.cloudlab.us/.
[2] [n.d.]. Intel DPDK: Data Plane Development Kit. https://www.dpdk.org.
[3] [n.d.]. P4 Behavioral Model. https://goo.gl/vzBLE4.
[4] [n.d.]. $P4_{16}$ language. https://goo.gl/wp6no2.
[5] [n.d.]. Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.
[6] 2007. Intel SSE4 Programming Reference. https://goo.gl/J4HkVo.
[7] 2014. Making Facebook's software infrastructure more energy efficient with Autoscale. https://goo.gl/692u64.
[8] 2017. A10. https://www.a10networks.com/.
[9] 2017. Data Sharing on traffic pattern inside Facebook's datacenter network. https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/.
[10] 2017. Loadbalancer.org Inc. https://www.loadbalancer.org/.
[11] 2017. Netscaler, Citrix Systems Inc. https://goo.gl/STMuUY.
[12] 2017. Nginx. https://www.nginx.com/.
[13] 2017. PI Library. https://goo.gl/8Np9HQ.
[14] 2019. Anonymous Source Code of the Concury Prototype. https://www.dropbox.com/s/ruou2l340uu1f4u/concury%20code.zip.
[15] 2019. Libtins network packet sniffing and crafting library. https://goo.gl/36qokt.
[16] 2019. Mininet. http://www.mininet.org/.
[17] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State. In *Proc. of USENIX NSDI*.
[18] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2009. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. of ACM SODA*.
[19] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *Proc. of ACM SIGCOMM*.
[20] Denis Charles and Kumar Chellapilla. 2008. Bloomier Filters: A Second Look. In *Proc. of ESA*.
[21] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA*. 30–39.
[22] David Chou et al. 2019. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proc. of ACM SOSP*.
[23] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*.
[24] Bin Fan, Dave Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*.
[25] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud scale load balancing with hardware and software. *Proc. of ACM SIGCOMM*.
[26] Xiaozhou Li, Dave Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys*.
[27] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. 1996. A Family of Perfect Hashing Methods. *Comput. J.* (1996).
[28] Rui Mao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*.
[29] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* (2008).
[30] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer. In *Proc. of USENIX NSDI*.
[31] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* (2004).
[32] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. *Proc. of ACM SIGCOMM*.
[33] B. Schlinker et al. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proc. of ACM SIGCOMM*.
[34] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016).
[35] Luis M. Vaquero and Luis Rodero-Merino. 2014. Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM CCR* (2014).
[36] R. Wang, D. Butnariu, and J. Rexford. 2011. Openflow-Based Server Load Balancing Gone Wild. *Proc. of ACM Hot-ICE*.
[37] Kok-Kiong Yap et al. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proc. of ACM SIGCOMM*.
[38] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. 2015. Fog Computing: Platform and Applications. In *Proc. of IEEE HotWeb*.
[39] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. 2017. Othello Hashing for Scalable and Fast Name Switching. In *Proc. of IEEE ICNP*.
[40] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. 2018. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking* (2018).
[41] Ye Yu, Xin Li, and Chen Qian. 2017. SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing. In *Proc. of ACM SIGCOMM Workshop on Mobile Edge Computing (MECCOM)*.