Re-designing Compact-structure based Forwarding for Programmable Networks

Shouqian Shi, Chen Qian, and Minmei Wang
Department of Computer Science and Engineering, University of California, Santa Cruz

***Abstract*—Forwarding packets based on networking names is essential for network protocols on different layers, where the 'names' could be addresses, packet/flow IDs, and content IDs. For long there have been efforts using dynamic and compact data structures for fast and memory-efficient forwarding. In this work, we identify that the recently developed programmable network paradigm has the potential to further reduce the time/memory complexity of forwarding structures by separating the data plane and control plane. This work presents the new designs of network forwarding structures under the programmable network paradigm, applying three typical dynamic and compact data structures: Bloom filters, Cuckoo hashing, and Othello hashing. We conduct careful analyses and experiments in real networks of these forwarding methods for multiple performance metrics, including lookup throughput, memory footprint, construction time, dynamic updates, and lookup errors. The results give rich insights on designing forwarding algorithms with dynamic and compact data structures. In particular, the new designs based on Cuckoo hashing and Othello hashing show significant advantages over the extensively studied Bloom filter based methods, in all situations discussed in this paper.**

I. Introduction

Packet forwarding is a fundamental function of various types of network devices running on different layers. For each incoming packet, the forwarding device transmits it to the link towards one of its neighbors, until reaching its destination. There are two main types of packet forwarding: 1) IP *prefix matching* that is mostly used on layer-3 routers; 2) *name-based matching* that is used on most other network devices. For name-based forwarding, the input of the forwarding algorithm is a *key* (also called as a *name* or *address* in different designs) included in the packet header, and the output is an entry that matches the key *exactly* and indicates an outgoing link. This work focuses on packet forwarding with such name-based matching, which attracts growing attentions in emerging network protocols and systems. We provide an incomplete list of recently proposed name-based forwarding designs:

1) On the link layer (layer-2 or L2), interconnected Ethernet has been used for large-scale data centers [22][54], enterprise networks[51][56], and metro-scale Ethernet [27][25][39][40], where the key is the MAC or other L2 addresses. Although many existing data centers use the fat tree based design that uses IP routing, name-based routing and forwarding still provides a number of advantages, including flexible management and host mobility. L2 name-based architectures are also suggested in many future network proposals [27][25][39].

978-1-7281-2700-2/19/$31.00 2019 © IEEE

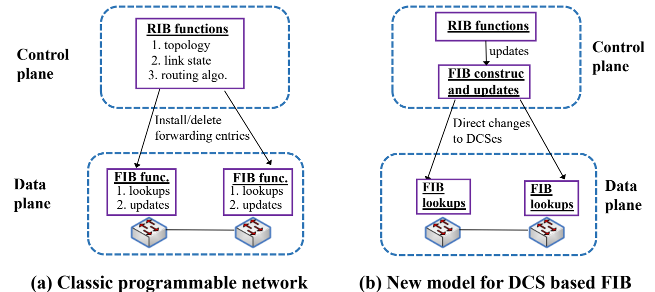


Fig. 1. Separating a DCS-based FIB to two planes

2) On the network layer (layer-3 or L3), flow-based networks, such as OpenFlow-style software defined networks (SDNs), match multiple fields in packet headers to perform fine-grained per-flow control on packet forwarding [56][23][38][43]. The matching key is some header fields. In addition, many new Internet architectures suggest flat-name forwarding in the network layer, such as MPLS [42], LTE [55], Mobility-first [41], and AIP [10].
3) On the application layer (layer-7 or L7), a content distribution network (CDN) uses the content ID as the key to search for the cache server that stores the content [20][30]. The emerging edge computing provides more sophisticated content/service caching services [45][50].

Unlike IP addresses, aggregating network names is tough – if ever possible. Large networks using name-based forwarding may suffer from the forwarding information base (FIB) *explosion problem*: a forwarding device needs to maintain a large number of key-action entries in the FIB. To resolve this problem, for long, there have been efforts to apply *dynamic and compact data structures* (DCSes) for the forwarding algorithms of network names, such as Bloom Filters [20][51][34], Cuckoo hashing [56][55], and Bloomier filters [16][15][52][53]. We summarize the desired properties of the DCSes for forwarding algorithm designs:

1) **Small memory footprint.** Fast memory is the most precious network resource, such as the on-chip memory (SRAM) on a switch, or the CPU cache on a server. DCSes reduce network infrastructure cost by using small memory.
2) **Fast lookups.** Faster lookups mean higher forwarding throughput. The throughput of a FIB should reach the line rate to avoid being a bottleneck.
3) **Dynamic updates.** Modern networks are highly dynamic due to massive incoming flows and host mobility. Hence, the DCSes should allow the FIB to be frequently updated.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

Although many FIB algorithms have been proposed, the recently developed programmable network paradigm [13][14], such as Software Defined Networks [21][35] and Network Functions Virtualization [7], still provides the potential to further reduce the time and memory complexity of forwarding algorithms. Hence there is a need of re-designing forwarding algorithms with the DCSes under this new paradigm. We design and implement **new forwarding algorithms** for programmable networks, by **re-visiting** three representative DCSes: Bloom Filters [12], Cuckoo hashing [36], and Bloomier filters (Othello hashing) [16][15][52].

As shown in Fig. 1(a), in a traditional design, the controller only runs the Routing Information Base (RIB), while the whole FIBs are stored in the data plane. The **key innovation** of our re-designs is show in Fig. 1(b). We relocate the memory and computation of the update function from many FIBs to the central control plane, and the data planes, while supporting direct updates, focus on fast lookups. Our approach significantly reduces data plane memory footprint while preserving the control plane scalability. We conduct careful analysis and experiments of the proposed methods for multiple performance metrics, including memory footprint, lookup throughput, construction time, dynamic updates, and lookup errors. The results can be utilized for future forwarding algorithm designs.

Our contributions are summarized as follows. 1) We propose a new design framework of FIBs in programmable networks. 2) We design new forwarding algorithms with DCSes in the programmable network paradigm that achieve small memory and high lookup throughput compared to all existing methods. 3) We implement the proposed methods in real network environments deployed in CloudLab [1] for real packet forwarding experiments. 4) Our results provide rich insights of designing forwarding algorithms. In particular, we find that the Bloom filter based methods, which have been extensively studied in the literature, are not ideal design choices compared to other proposed methods, in all situations studied in this paper.

The balance of this paper is organized as follows. § II presents the related work and three DCSes. § III introduces the network models. We present the forwarding algorithm designs in § IV and provide the analysis results in § V. The evaluation results are shown in § VII. We discuss the insights of this study in § VIII and conclude this work in § IX.

II. Background and Related Work

To address the FIB explosion problem, DCSes have been proposed as the forwarding data structures in various types of network devices.

Bloom filters. The Bloom filter [12] is one of the most popular DCSes used in network protocols. A filter data structure is a brief expression of a set of keys K. By querying a key k, a filter should return *True* if $k \in K$ or *False* otherwise. A well-known feature of Bloom filters is that its results include false positives but no false negatives. The basic idea of using Bloom filters for FIBs is that for every link to a neighbor, the forwarding node maintains a Bloom filter for the set of

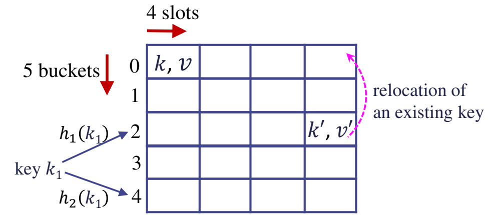


Fig. 2. (2,4)-Cuckoo hashing table

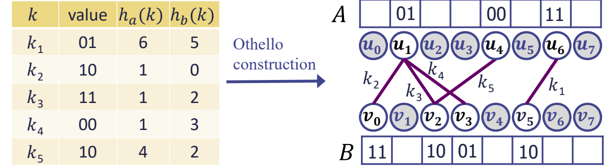


Fig. 3. Construction of Othello hashing

names/addresses that should be forwarded to this link, such as Summary Cache [20] and BUFFALO [51]. Each lookup takes $O(K \cdot d)$ time, and each update takes $O(d + K)$ time, where d is the node degree. Despite the complex lookup and update, false positives still occur, which hurt the bandwidth. The Shifting Bloom Filter [48] achieves fast lookups, but its false positive rate is high.

Cuckoo hashing. Cuckoo hashing [36] is a key-value mapping data structure that achieves $O(1)$ lookup time in the worst case and $O(1)$ update time on average. Many recent system designs choose the (2,4)-Cuckoo hashing [18][56][55] to maximize the memory load factor. As shown in Fig. 2: a Cuckoo hashing is a table of a number of buckets, and each bucket has 4 slots. Every key-value pair is stored in one of the 8 slots of the two possible buckets based on the two hash function results $h_1(k)$ and $h_2(k)$. The lookup for the value of a key k is to fetch the two buckets and match k with the keys from all the 8 slots. The value associated with the matched key is the result. FIBs using Cuckoo hashing store the link or port index in each 'value' field together with the key (name), such as CuckooSwitch [56]. ScaleBricks [55] uses both Cuckoo hashing and SetSep [19] for cluster network functions. SetSep is a compact structure with no update function, and hence it is out of the scope of this work.

Bloomier filters. The Bloomier filters [16][15] and their variants Othello hashing [52][47] and Coloring Embedder [49] are key-value lookup tables inspired by dynamic perfect hashing [31][11]. We use Othello hashing as an example to introduce this idea. As shown in Fig. 3, Othello builds an acyclic bipartite graph G, where every key k corresponds to an edge in the bipartite graph connecting the $h_a(k)$-th vertex on the top and the $h_b(k)$-th vertex on the bottom, based on its two hash functions h_a and h_b. As shown in Fig. 4(a), the Othello lookup result is simply the value of $A[h_1(k)] \bigoplus B[h_2(k)]$, where A and B are two arrays computed from G and the key-value information. If G is acyclic, the values in A and B can be easily determined to satisfy such lookup operation. Rehashing will happen if a cycle is found in G. The $O(1)$ update time is proved in [52]. The important features of Othello are 1) the memory cost is small as it stores no keys in the lookup

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

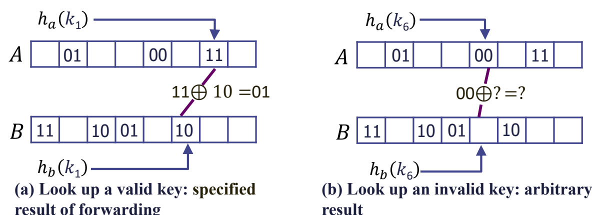


Fig. 4. Lookup of Othello hashing

structure; 2) it uses only two memory accesses to lookup a key in the worst case; 3) it takes $O(1)$ average time for each addition/deletion/update. Concise [52] is an L2 FIB design based on Othello. One weakness of Concise is that it cannot tell whether a key (name/address) exists in the network. If a packet carries an invalid key, Concise forwards it to an arbitrary neighbor, as shown in Fig. 4(b). SDLB [53] and Concury [44] are L4 load balancers using Othello.

III. Network models

A. Optimizing DCSes in Programmable Networks

Programmable networks use software, running on general-purpose computers or programmable switches, to perform various network functions, e.g., packet forwarding, firewalls, load balancer, and traffic monitoring. The typical examples include SDNs [32][26][24][21][35][13], software routing and switching [28][38][56][23][43], and network functions virtualization (NFV) [7][37][17]. We observe that the programmable network paradigm provides *a new opportunity* to allow new data structures and algorithms being run on network devices and their further optimizations. As shown in Fig. 1(a), existing networks require each data plane device to host the entire FIB, such as the flow table, which supports both the lookup and update functions. Even in the current SDNs, the controller only runs the Routing Information Base (RIB) but not the FIBs. We propose to split the FIB into two components, which perform the lookup and update functions respectively. The **FIB data plane (DP) component** focuses on the lookup and only performs simple memory writes for updates. Hence the DP fits in fast memory (e.g., switch ASICs or CPU cache). The **FIB control plane (CP) component** is responsible for the full states and calculations for the construction and updates of the DPs and can be run on a server. This idea creates two optimization opportunities for FIB designs: 1) without the update component, the FIB lookup function can be built with a DCS with small memory footprint; 2) The FIB construction and update component can be reused for network-wide data-plane nodes, to preserve control-plane scalability. The reusability depends on the specific application. However, it is always more efficient than maintaining a different update component for every node.

B. Forwarding Model

There are a total number of n forwarding nodes interconnected to form a network. Each port of a node is linked to either a host or another node. The number of ports of a node is d. Each host has an address k as its unique ID or search key. The forwarding structure (i.e., the FIB) on each node should include the host-port mapping of *all* hosts in the network. The port information of k in the FIB indicates the next-hop node to route the packet to host k.

Practical examples: In a large-scale Ethernet (e.g., large organizations, metro Ethernet, and L2 data center networks)[27][22][39], there are a huge amount of physical hosts. Each host has an ID (e.g., its MAC address). An interconnection of switches connects the hosts. Each switch has multiple ports connecting neighboring switches and hosts. A switch may be a **gateway** that connects to external networks and filters alien keys – an alien key is a key that does not exist in the network – or a core switch that only connects to internal devices. Each network packet (Ethernet frame) processed by a switch includes the MAC of the destination host. A switch forwards the packet to a neighbor based on FIB lookups using the MAC. Many modern networks are variants of this model [27][22][39]. For flow-based networks [32], the flow ID may be a combination of source/destination IPs, MACs, and other header fields. The forwarding may be per flow basis, rather than per destination basis. LTE backhaul networks and core networks can also be regarded as an instance of the L2 network model, especially for the down streams from the Internet to mobile phones. The destinations are mobile phones, and the IDs are Tunnel End Point Identifiers (TEIDs) or International Mobile Equipment Identities (IMEIs) of the mobiles [55]. We do not study routing protocols in this work and focus on forwarding.

Application-layer (L7) forwarding. L7 network forwarding model is slightly different from the above L2/L3 models. The examples of L7 forwarding include CDN content lookups [20], distributed data storage [9], P2P systems, and edge computing [45]. In L7, a node can be connected to arbitrarily many neighbors, because those connection links are virtual, such as TCP sessions. The number of neighbors of an L2 switch is bounded by the number of physical ports: an important parameter of the switch related to its price.

Packet forwarding in L2 and L7 can be simplified and unified in the following statement: given a packet carrying the key k, the forwarding structure should return the index of the corresponding outgoing link. The network updates discussed here can be *key addition* (new host joining with a new address in L2 and new content being stored in L7), *key deletion* (existing host failing or leaving in L2, and new content deletion in L7), or *value update of a key* (host moving to a new location or routing path changes in L2 and content being stored at a new location in L7).

Although this work mainly focuses on the L2/L3 forwarding model due to space limit, the proposed methods and results may still apply to L7.

IV. Forwarding structure designs

By exploring the potential of the programmable network paradigm, we optimize the lookup/memory/update efficiency of DCSes based forwarding algorithms. We propose three forwarding structures and algorithms: Bloom Forwarder (BFW) based on Bloom filters [12], Cuckoo Forwarder (CFW) based on a new data structure Cuckoo Filtable, and Othello Forwarder (OFW) that extends Othello hashing [52]. **CFW and**

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

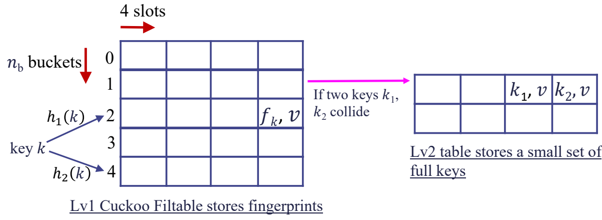
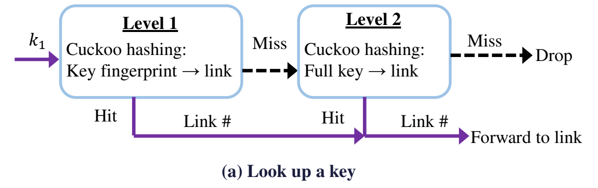


Fig. 5. Example of Cuckoo Filtable



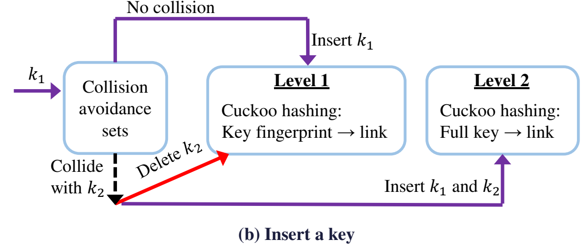


Fig. 6. CFW insertion and lookup workflows

OFW are considered our main design contributions, and BFW is a baseline for comparison.

A. Bloom Forwarder (BFW)

Limitations of existing methods. Both BUFFALO [51] and Summary Cache [20] use Bloom based forwarding, and their ideas are similar. For every outgoing link, the forwarding node maintains a Bloom filter (BF) representing the keys of the packets that should be forwarded to this link. To look up a key, the node iteratively checks each BF and then picks the index of the first matched BF [20] as the link index. There are d BFs for d links on a node. Summary Cache uses the counting Bloom filters (CBFs), which support deletion operations. The drawback of CBFs is that they increase the memory cost by a factor of $\log_2 n_k$ in the worst case, where n_k is the number of keys. BUFFALO [51] uses BFs as its DP and maintains CBFs in its CP to save the switch ASICs. The main weakness of using CBFs in the CP is that CBFs only record the hashed bits but do not store keys. Hence, it is impossible to reconstruct the DP in cases like topology changes and BF resizing because reconstruction requires retrieving all original keys to build new BFs.

Bloom Forwarder (BFW). BFW uses a similar DP design to BUFFALO [51], but a different CP design. We follow the extensive optimizations proposed in BUFFALO to minimize the false positives. In addition, we propose to use a Cuckoo hashing table to store all keys in the CP because the CBFs do not support DP reconstruction. The DP includes both the BFs of all ports for lookups and a set of CBFs to support incremental updates without reconstruction. The CBFs are kept in DP because a centralized CP may neither have enough memory to maintain all CBFs of all forwarding nodes nor enough computation power to perform a small update (such as new address join or leave), which will trigger different updates in different CBFs for all DPs.

B. Cuckoo Forwarder (CFW)

Limitations of existing methods. CuckooSwitch [56], the typical example of Cuckoo hashing based forwarding, uses the (2,4)-Cuckoo hashing table as its FIB, and stores the full keys in the hash table. This approach incurs high memory overhead on the DP. Cuckoo filter [18] stores the fingerprints of keys rather than the full keys, but it only supports membership queries and cannot be used as the FIB.

New Design: Cuckoo Forwarder (CFW) Data Plane. The CFW DP uses a **new structure design** proposed by us called Cuckoo Filtable, which borrows the ideas from both Cuckoo hashing and Cuckoo filters. It is a table of n_b buckets and each bucket includes 4 slots. Each slot stores the *fingerprint* f_k of a key k and the value v associated with k, which is the index of the link to forward packets carrying ID k, as shown in Fig. 5. f_k is the fingerprint of k with a fixed and much shorter length than k, which can be computed by applying a hash function to k. Storing f_k instead of k significantly reduces the memory cost. To lookup k for an incoming packet, CFW fetches two buckets based on $h_1(k)$ and $h_2(k)$ and computes the fingerprint f_k. Then for each slot in these two buckets, CFW compares f_k with the stored fingerprint. If there is a match, the stored value v, which is the link index of the next hop neighbor, will be returned. Different from the existing "partial key Cuckoo" solution [29], the Cuckoo Filtable addresses the following challenges.

Challenges in CFW DP design. By storing the fingerprints, CFW experiences the *false hits*: The fingerprint $f(k)$ of a key k will match a slot that stores the fingerprint of another key k' if $f(k) = f(k')$. There are two kinds of false hits. **1)** k does not exist in the network, called an *alien key*, and it has the same fingerprint as an existing key k'. This type of false hits is called *false positives*. It is impossible to avoid false positives unless CFW stores the entire keys. The false positive rate depends on the length of the fingerprints. **2)** k and k' both exist in the network and happen to share the same fingerprint and locate in the same bucket. This is called a *valid key collision*. This problem is critical: in an L2 Ethernet-based enterprise or data center network, all forwarding nodes in a subnet/data center may share the same set of keys [27], and thus, a pair of colliding valid keys k and k' will collide at *every* node. One of the destinations will never be successfully accessed. We call this problem as *key shadowing*.

To resolve valid key collisions, we adopt a two-level design, as shown in Fig. 5. Level 1 is a Cuckoo Filtable that stores non-colliding fingerprints and their values, as described above. Level 2 is a Cuckoo hashing table that stores full keys whose fingerprints collide with one or more other keys. A key k will be moved to Level 2 if these two conditions are satisfied: 1) there is another k' such that $f_k = f_{k'}$; and 2) k and k' have at least one common bucket. Each key relocated to Level 2 will be inserted to a *collision avoidance set* (explained later) to prevent future false hits. We expect that only a small portion of keys are stored in Level 2. Thus the memory cost does not

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

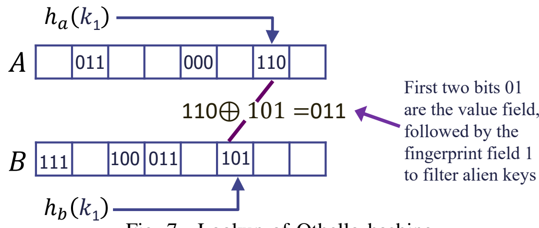


Fig. 7. Lookup of Othello hashing

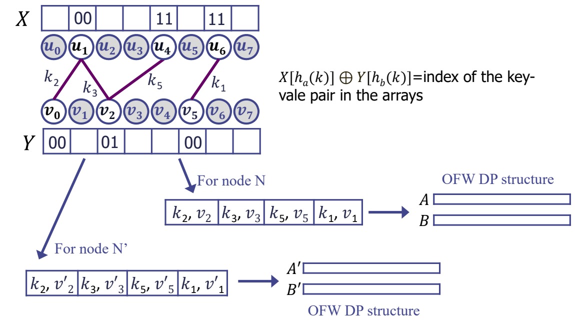


Fig. 8. OthelloSet for network-wide updates

increase significantly. A lookup operation is to first search for the fingerprint in the first level, and if there is a miss, CFW looks up the key in the second level, as shown in Fig. 6(a).

New Design: CFW Control Plane. The CFW CP stores the network topology and routing information. For the FIB, the CFW CP uses a two-level design to support fast constructions and updates for all DPs. The difference between the CP FIB and the DP FIB is that each slot in Level 1 of CP FIB stores three fields: the full key k, the fingerprint f_k, and the physical host ID (only for multi-tenant networks). The full keys are used to help key additions and deletions. When a reconstruction is needed, the two-level CP FIB can immediately be converted to the two-level DP FIB by removing all full key fields. Besides, when some DPs hold the same set of keys, the CP FIBs of the nodes share the same 'skeleton': the same key at different nodes is in the same position of the lookup structure (though their value fields are different). The construction of a DP FIB is to directly copy the skeleton without full keys and resolve each key to the port index on the node based on the routing information stored in the RIB. **This property has been not explored by any prior work.** Based on this design, if there is a central control program, a network-wide update will be extremely fast.

Challenges in CFW CP design. One problem may happen when there is a three-key collision: keys a, b, and c have the same fingerprint and share a bucket. a and b are already stored in the Level 2 hash table. When c is added to the FIB, it will be directly added to the Level 1 without collisions – because a and b are not there. However, it causes a problem in DP lookups: all lookups of a and b will hit c's slot.

In CFW, this problem is resolved by storing additional information in Level 1 of CP FIB. Level 1 maintains a *collision avoidance set* at each bucket, which stores all valid keys that have this bucket as one of its two alternative buckets. For every key being inserted, CFW should first check if its fingerprint collides with any fingerprint in the collision avoidance sets of its two alternative buckets to avoid possible collisions, as shown in Fig. 6(b). Every inserted key is added to the collision avoidance set of both the alternative buckets. If a collision is detected before the insertion, the two colliding keys are moved to the second level.

C. Othello Forwarder (OFW)

We further explore the efficiency of Othello hashing [52] for a new FIB design in programmable networks.

Limitations of existing methods. Concise [52] is an L2 FIB based on Othello hashing. Concise has two main limitations. 1) It only includes the design for a single switch but misses the design for network-wide CP-DP coordination; 2) It has no ability to filter alien keys that do not belong to the network.

New Design: Othello Forwarder (OFW) Data Plane. The OFW DP consists of two arrays A and B, and two different hash functions h_1 and h_2. The lookup of a given key k works as follows: the $h_1(k)$-th element in A, $A[h_1(k)]$, and the $h_2(k)$-th element in B, $B[h_2(k)]$, are fetched; and the DP calculates $\tau = A[h_1(k)] \bigoplus B[h_2(k)]$. The result τ is the concatenation $v||f$, where v is the index of the forwarding port and f is a fingerprint to filter alien keys, as in Fig. 7. The DP then calculates the fingerprint f_k of the key k. If $f_k = f$, v is returned, otherwise *null* is returned to indicate an alien key. If all the request keys are guaranteed to be valid, length of f is set to 0. The main drawback of adding fingerprints is the high memory cost because the total number of slots in arrays A and B is $2.33n_k$ for n_k keys, and thus, one bit in the fingerprint field contributes 2.33 bits to the overall memory. Intuitively, when the fingerprint grows by 1 bit, the DP can reduce 50% false hits. An interesting result is that using only 1 bit can filter more than 50% alien keys: we call the last bit of a fingerprint as 'emptiness indicator', which is set to 1 when this element in the array A or B is associated with one or more keys. If both indicators in the fetched two elements are 1, then there may be a key matching the two values. If either of them is 0, then the lookup key must be an alien key.

New Design: OFW Control Plane. The OFW CP uses a new data structure, OthelloSet, to support efficient network-wide FIB updates. The simplest way to maintain the OFW CP is to maintain an array of key-port pairs for every node. In this way, however, the CP may not have enough memory to hold all the arrays and the DP construction will be prohibitively time-consuming. Our important observation is that if some nodes share the same set of keys, they also share the same bipartite graph G because G only depends on the keys. Hence, OFW CP maintains the routing information, an array of key-host pairs, and a 'skeleton' for all DPs. As shown in Fig. 8, one graph G and two arrays X and Y are maintained in the CP as the 'skeleton', such that $X[h_a(k)] \bigoplus Y[h_b(k)]$ is the index of the key-host pair array. For construction of each node, OFW CP calculates the host-port mapping for each key in the array. Then based on G and the derived key-port mapping, the OFW DP (arrays A and B for lookup) can be easily constructed.

The reasons to use OthelloSet are: 1) OthelloSet stores full key-value information which suffices to be a CP data structure; 2) we have to maintain a bipartite graph G of the Othellos at the CP to quickly synchronize with DPs. The key insight here

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

Symbol	Description
n_k	total number of valid keys
d	number of links
l_p	length of port index encoding
l_f	length of fingerprint field in slots
l_k	length of a key
l_s	length of counters in CBF
l_b	bucket length in Cuckoo hashing or Cuckoo filter
n_b	number of buckets a key is mapped to in Cuckoo hashing or Cuckoo Filtable (usually 2)
n_s	number of a slots in a bucket (usually 4)
r_l	load factor of Cuckoo hashing or Cuckoo filter
e_l	$e_l = \frac{1}{r_l}$
l	number of hash functions for a Bloom filter

TABLE I
NOTATIONS

is that, although the link indices (values) corresponding to the keys are different on different nodes, h_1, h_2, and G can be shared between the CP OthelloSet and all CPs in the network. To construct a new FIB or to incrementally update FIBs in the network, Othello reconstruction is no longer needed, the CP only has to determine the values to fill the slots in arrays A and B.

D. Control plane reusing and scalability

The key reason of letting the BFW DP to store the CBFs is that the CBFs cannot be reused among different forwarding nodes, and every node must have a set of unique CBFs. Hence, storing the CBFs in the central controller will cause significant scalability problem. In fact, the server used in our experiments cannot afford storing CBFs for over 100 nodes. On the other hand, the DPs of CFW and OFW can be reused if different nodes are forwarding a same set of addresses, which is true in many L2 networks [22][27][25][39]. We understand that in some practical networks that are not pure L2 flat networks, nodes in different regions may have different sets of addresses. However, the designs of CFW and OFW can still significantly reduce the control plane overhead if some nodes share similar sets of addresses, e.g., those in a same subnet.

V. ANALYSIS AND FURTHER OPTIMIZATION

We conduct theoretical analysis on the following three aspects: the memory footprint in the DP, times of hash function invocations and memory reads for each lookup, and times of hash function invocations and memory reads and writes for each FIB update. We also present the system design details guided by the analysis. The notations are listed in Table I.

A. DP memory footprint

The data structures at the DPs are analyzed as two parts for BFW and CFW – the total memory footprint and the memory footprint of frequently accessed parts during lookup. We use the symbol M to denote the overall memory footprint, and let M_f to be the memory footprint of the mostly accessed parts that can be hosted in fast memory.

BFW. For BFW, a FIB is divided into two parts: the counting Bloom filter and the Bloom filter. The Bloom filter is the frequently accessed part. The FIB memory footprint of BFW M^b and M^b_f (both in bits) are $(1 + l_s)m$ and m, respectively, where m is the sum of the lengths of all Bloom filters and $m = hn_k/\ln 2$ for h hash functions [12].

CFW. The CFW DP consists of two levels. Level 1 is the Cuckoo Filtable that stores key fingerprints, which is the frequently accessed part. Level 2 stores the full keys for the colliding keys. We first calculate the expected portion of keys at Level 1 η and then derive the expected CFW memory footprint M^c. Our experiments show that η is a function of l_f and is independent of n_k. We define the function $E_\eta(l)$ to reflect the experimental results. Based on that, the memory footprint of Level 1 is $M^c_f = E_\eta(l_f) \cdot n_k \cdot e_l(l_f + l_p)$ and the total memory is $M^c = M^c_f + (1 - E_\eta(l_f))n_k \cdot e_l(l_k + l_p)$.

OFW. There is only one data structure in the OFW DP, which means the whole FIB memory M^o and the most accessed memory M^o_f are the same: $M^o_f = M^o = 2.33n_k \cdot (l_p + l_f)$, where the coefficient 2.33 is derived in [52].

B. Time complexity

Although different FIB designs have different workflows in lookup, hashing keys and loading memory contents are common and most time consuming, compared to other operations such as calculating memory offsets. Hence, we use the number of memory accesses and hash function invocations to measure time complexity. We denote the numbers of memory accesses and hash function invocations as C_m and C_h respectively. We denote the expected numbers of memory loads and hash function invocations of an alien key as $C_{m,e}$ and $C_{h,e}$ respectively. The detailed derivations of the following results are skipped due to space limit, but are available on [6].

BFW.

$$E(C^b_h) = \sum_{i=1}^{d-1} \frac{1}{d}\left((i-1)C_t + l\right) = \frac{d-1}{2}C_t + l$$
$$E(C^b_{h,e}) = \left(\sum_{i=1}^{d-1} \left((1-p)^l\right)^{i-1} \cdot \left(1-(1-p)^l\right)(i \cdot C_t)\right) + \left((1-p)^l\right)^d (d \cdot C_t) \tag{1}$$

$$\begin{aligned} E(C^b_m) &= E(\lceil l_k/l_c \rceil + C^b_h) = \lceil l_k/l_c \rceil + E(C^b_h) \\ E(C^b_{m,e}) &= E(\lceil l_k/l_c \rceil + C^b_{h,e}) = \lceil l_k/l_c \rceil + E(C^b_{h,e}) \end{aligned} \tag{2}$$

CFW. (Assuming the key locations are uniformly random)

$$\begin{aligned} E(C^c_h) &= 2 + \frac{n_b - 1}{2} + (1 - E_\eta(l_f))(1 + n_b) \\ E(C^c_{h,e}) &= 1 + n_b + n_b = 1 + 2n_b \end{aligned} \tag{3}$$

$$\begin{aligned} E(C^c_m) &= \lceil l_k/l_c \rceil + \sum_{i=1}^{n_b \cdot n_s} \frac{\lfloor i/n_s \rfloor \cdot E(C_b) + E(C_{m,i})}{n_b \cdot n_s} \\ E(C^c_{m,e}) &= \lceil l_k/l_c \rceil + n_b \cdot E(C_b) \end{aligned} \tag{4}$$

OFW. The expected portion of empty slots in A and B are: $\epsilon_a = (\frac{m_a-1}{m_a})^{n_k} \approx e^{-\frac{n_k}{m_a}} \approx 0.471$ and $\epsilon_b = (\frac{m_b-1}{m_b})^{n_k} \approx e^{-\frac{n_k}{m_b}} \approx 0.368$. Let $l_g = gcd(l_f + l_p, l_c)$. Assume the $l_f + l_p$ is always smaller than l_c. We get:

$$\begin{aligned} C^o_h &= 3 \\ C^o_{h,e} &= \epsilon_a + 2 \cdot (1 - \epsilon_a)\epsilon_b + 3 \cdot (1 - \epsilon_a)(1 - \epsilon_b) \end{aligned} \tag{5}$$

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

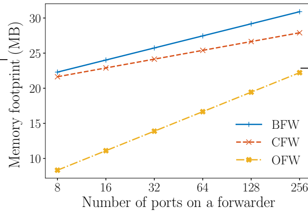


Fig. 9. Memory with gateways

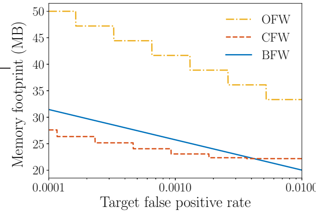


Fig. 10. Memory w/o gateways

$$E(C^o_m) = \lceil l_k/l_c \rceil + 2 \cdot \left(1 + \frac{l_f + l_p - l_g}{l_c}\right)$$
$$E(C^o_{m,e}) = \lceil l_k/l_c \rceil + ((1-\epsilon_a)\epsilon_b + 2 \cdot (1-\epsilon_a)(1-\epsilon_b)) \cdot \left(1 + \frac{l_f + l_p - l_g}{l_c}\right) \quad (6)$$

C. Collision rate and false positive rate

We consider two problems caused by hash collisions: valid key collisions and false positives. A key collision happens between two valid keys, causing the lookups of the two keys ending up with the same value. A false positive happens when an alien key that does not exist in the network gets the value of an existing key. We use *CR* and *FP* to denote the **valid key collision rate per lookup** and the **alien key false positive rate per lookup**, respectively. We obtain the following results.

BFW.

$$FP^b = 1 - \left(1 - (1-p)^l\right)^d$$
$$E(CR^b) = \sum_{i=1}^{d} \left(1 - \frac{1}{2^l}\right)^{i-1} \cdot \frac{1}{d2^l} \quad (7)$$

CFW.

$$CR^c = 0$$
$$FP^c = 1 - (1 - \frac{1}{2^{l_f}})^{r_l E_\eta(l_f) \cdot n_s n_b} \quad (8)$$

OFW.

$$CR^o = 0$$
$$FP^o = \frac{1}{2^{l_f - 1}} \cdot (1-\epsilon_a) \cdot (1-\epsilon_b) \quad (9)$$

D. Numerical results and discussions

We show the numerical results to compare different forwarders and make some design choices based on the numerical results.

DP memory footprint. We consider the DP memory in two situations: with and without gateways. As described in § III, the gateways may exist at the border of a network. A gateway is also a forwarder, but with full key information. Hence, a gateway will drop invalid requests and only forward valid requests. Having all incoming requests passing through the gateways, there is no false positive at internal forwarders. If gateways are used, then other forwarding nodes do not need to filter alien keys. Note even if gateways exist, BFWs on internal nodes still suffer from the key collisions, but there is no collision in CFW and OFW. We fix n_k to be $10M$, $l_k = 128$, $CR = 1‰$ for BFW, $l_f = 0$ for OFW, and we pick l_f for CFW giving out the smallest memory footprint. We let l_p range from 5 to 13. The results in Fig. 9 show that OFW provides the least memory cost, around 20%-60% of the other two.

If there is no gateway, we calculate the smallest memory footprints of the three forwarders achieving a certain level of false positive rate. We fix n_k to $10M$ and let the false positive rate range from 0.01 to 0.0001. We carefully adjust the parameters of the three forwarders to let them have the smallest memory footprint while meeting the target false positive rate. The numerical results are shown in Fig. 10.

Comparing the results in Figures 9 and 10, it is clear that when gateways exist, OFW costs the much less memory than the other two designs. However, without gateways, OFW needs much more memory to achieve a certain level of false positive. CFW costs the least memory when false positive $< 0.4\%$. Hence, an ideal solution may be using Cuckoo hashing or OthelloSet at the gateways and using OFWs for the remaining internal nodes.

VI. Implementation

Algorithm implementation. We implement all three forwarder prototypes in a total of 4360 lines of C++ code and these prototypes share a part of the code. We build the CFW prototype based on the *presized_cuckoo_map* implementation in Tensorflow repository [8], with several major modifications to implement Cuckoo Filtable and the control plane of CFW. We also implement the collision avoidance sets at the control plane Level 1 table. The insertion workflow is specially implemented and tested for the two-level Cuckoo Filtable and the collision avoidance sets. We reuse the code from the GitHub repository of Othello hashing [3] and add the extra functions such as fingerprint checking and the control plane to data plane incremental synchronization. As the Bloom filters and CBFs are easy to implement, we just implement the BFW and its control plane from scratch and implement the incremental update feature. We adopt the Google FarmHash [2] as the hash function for all experiments.

Algorithm benchmark setup. We evaluate the single-thread performance of three forwarder algorithms 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).

CloudLab benchmark setup. We implement the forwarder prototypes BFW, CFW, and OFW using Intel Data Plane Development Kit (DPDK) [4] running in CloudLab [1]. DPDK is a series of libraries for fast user-space packet processing [4]. DPDK is useful for bypassing the complex networking stack in the Linux kernel, and it has the utility functions for huge-page memory allocation and lockless FIFO, etc. CloudLab [1] is a research infrastructure to host experiments for real networks and systems. 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 the forwarder prototypes. Each of the two nodes is equipped with one Dual-port Intel X520 10Gbps NIC, with 8 lanes of PCIe V3.0 connections between the CPU and the NIC. Each node has two Intel E5-

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

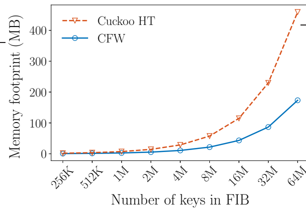


Fig. 11. Memory: CFW vs. Cuckoo hashing

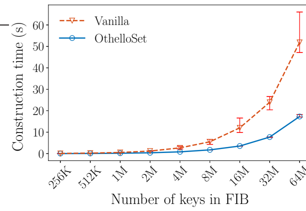


Fig. 12. DP construction: OthelloSet vs. Concise

2660 v3 10-core CPUs at 2.60 GHz. The Ethernet connection between the two nodes is 2x10Gbps. The switches between the two nodes support OpenFlow [32] and provide the full bandwidth.

Logically, Node 1 works as one of the forwarders in the network, and Node 2 works as all other nodes in the network, including gateways and switches. Node 2 uses the DPDK official packet generator Pktgen-DPDK [5] to generate random packets and sends them to Node 1. The destination IDs carried by the generated packets are uniformly sampled from a set of valid IDs. BFW, CFW, or OFW is deployed on Node 1 and forwards each packet back to Node 2 after determining the outbound port of the packet. By specifying a virtual link between the two servers, CloudLab configures the OpenFlow switches such that all packets from Node 1, with different destination IDs, will be received by Node 2. Node 2 then records the receiving bandwidth as the throughput of the whole system.

Network setup. We use one ISP network topology from the Rocketfuel project [46] as the topology model of the simulated network. Gateways are placed in the networks. l_d of OFW is set to 0, while l_d of Level 1 of CFW is set to 13 for lowest memory footprint. The lookup keys may be valid or alien, sampled from the following categories: 32-bit IPv4 addresses, 48-bit MAC addresses, 128-bit IPv6 addresses, and 104-bit 5-tuples.

VII. Evaluation

In this section, we carry out the algorithm benchmark and the CloudLab experiments to evaluate the performance of the three forwarder prototypes.

A. Comparison methodology

The distributions of lookup requests are simulated in two types: uniform distribution and Zipfian distribution. To understand the performance variations, each data point is the average of 10 experiments with different random seeds and the error bar on each data point shows the minimum and maximum value among the 10 results.

We conduct two kinds of comparisons: 1) Algorithm micro-benchmarks to examine different performance metrics; 2) Real packet forwarding experiments in CloudLab to understand the overall performances of the three forwarders in a real network. For algorithm micro-benchmarks, we compare the following performance metrics of all three forwarders: 1) Lookup throughput of valid and invalid addresses; 2) Control plane to data plane synchronization latency; 3) Control plane construction time.

B. Algorithm evaluation

Compare to prior methods. We have conducted the experiments of the studied methods with prior solutions: CFW vs. CuckooSwitch [56]; OFW vs. Concise [52]. **All of BFW, CFW, and OFW have a better or same performance in throughput and memory efficiency compared to prior solutions**. We show some representative results. We calculate the memory footprint for a single FIB to show that the CFW saves a considerable amount of memory compared to Cuckoo hashing as in CuckooSwitch. We set the $l_k = 64$ (MAC addresses) for both FIBs, and $FP = 1‰$ for Cuckoo Filtable. The results in Fig. 11 shows CFW, avoiding storing full keys, saves $> 3x$ memory compared to Cuckoo hashing. To show the advantage of adopting OthelloSet in OFW, we compare the construction time for a single forwarder: exporting OFW DP from OthelloSet CP skeleton vs. building OFW DP from scratch. We set $l_k = 48$ and $l_p = 8$. As shown in Fig. 12, OthelloSet achieves $>$ 3 faster DP construction and for a network of 64M entries. In summary, both CFW and OFW significantly improve the existing methods. We show more results by comparing them with BFW.

Lookup throughput. We evaluate the lookup throughput of both the gateway node and the core node. Figures 13 to 16 show the throughput of BFW, BFW gateway (BGW), CFW, CFW gateway (CGW), OFW, and OFW gateway (OGW) in a network where forwarding addresses are valid MAC addresses. The experiments are performed with single-thread instances of the three prototypes. We change the total amount of addresses stored in the FIB and observe the throughput in terms of million queries per second (Mqps).

The throughput decreases with the growth of FIB size because larger FIBs incur higher cache miss rates. OFW performs around 3x faster than CFW because of its small memory and simple lookup logic. BFW performs >10x worse than the other two. OGW performs 2x faster compared to other gateways when FIB size is small. As memory loads dominant the lookup latency for gateways when FIB is large, the lookup throughputs of all three forwarders are close. OFW performs slightly better under Zipfian distribution than under uniform distribution when the FIB size is 4M.

Different types of keys. We evaluate the lookup throughput for different key types, including IPv4, MAC, IPv6, flow ID, and URL (CDN content name). The results in Fig. 17 show that OFW always achieves the highest throughput, seconded by CFW.

Alien addresses. To understand the difference between lookups of alien addresses and valid addresses, we also examine the alien address lookup at gateways. Fig. 18 shows the throughput of BFW gateway (BGW), CFW gateway (CGW), and OFW gateway (OGW) where forwarding addresses are invalid MAC addresses. We vary the total amount of addresses stored in the FIBs. All gateways show performance decreases with alien addresses because CFW perform key matching for all addresses in the two buckets of the two levels (16 slots in total) to conclude the address is alien, and OFW performs one extra address lookup to detect the alien address.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

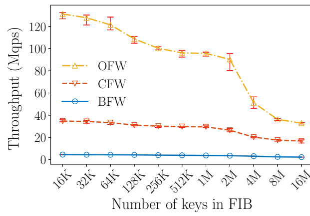


Fig. 13. DP lookup throughput for Zipfian distribution

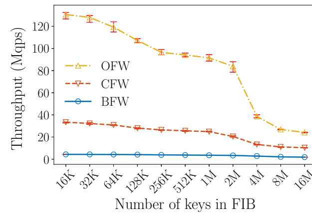


Fig. 14. DP lookup throughput for uniform distribution

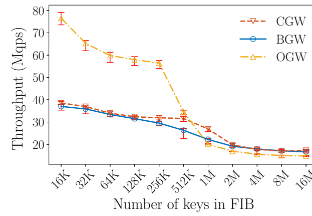


Fig. 15. Gateway throughput for Zipfian distribution

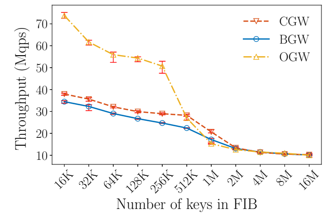


Fig. 16. Gateway throughput for uniform distribution

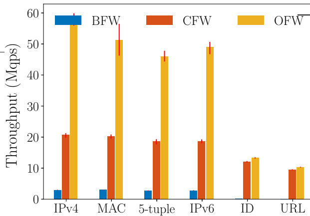


Fig. 17. Lookup throughput for different key types

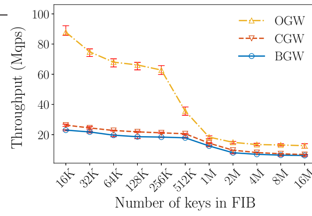


Fig. 18. Gateway throughput for invalid addresses

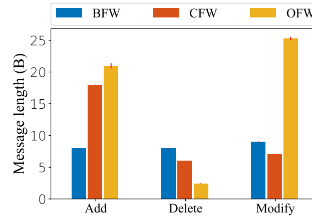


Fig. 19. Length of incremental update messages

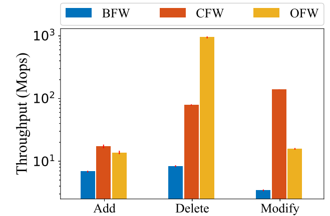


Fig. 20. Throughput (speed) of incremental updates

The performance drop for OFW is caused by the memory expansion, and the decrease only happens at small FIB sizes.

Data plane incremental update. As the valid addresses and their corresponding values are subject to change at runtime to reflect the network dynamics, FIB incremental updates happen frequently. The workflow of an incremental update is modeled as below. 1) The control plane receives an update report from the application specific message sources. Updates have three types: key addition, key deletion, and value modification. 2) The control plane updates the FIB skeleton to reflect the change and generates update messages for data planes based on the skeleton and the network routing information. 3) The data plane of each node receives the update message and updates its FIB accordingly.

The evaluation focuses on the communication overhead between the control plane and data planes, as well as the update throughput for the data planes. We set the FIB size to 4M and use the MAC addresses. We uniformly generate update messages of three different types and apply the same sequence of updates to the three forwarders. We record the average message lengths and the finish time of different update types, and we calculate the throughput of different update types in millions of operations per second (Mops).

Fig. 19 shows the update message lengths of BFW, CFW, and OFW. Value modification messages of OFW is longer than those of CFW because a value modification in OFW involves recoloring the whole connected component. Deletion messages are much shorter for OFW because it only needs to mark the empty indicator bits in up to 2 slots. Though CFW and OFW do not need to include full keys in the update messages, their addition messages are longer because CFW needs to include the cuckoo path, and the OFW needs to include the recoloring.

Fig. 20 shows the update throughput of BFW, CFW, and OFW. OFW is fast in key deletion because it only needs to mark the empty indicator bits. CFW is more than 10 times faster than others on value modifications because the update of CFW is simply copying the value to the specified slot. As we expect the update is less than 1M per second, all the three forwarders support realtime incremental updates.

Construction time. Although most updates in a network are incremental updates, there are always cases where new DP construction is needed, such as system checkpoint loading or forwarding node addition. We examine the construction time of a forwarding structure. Fig. 21 shows the control plane construction time at different FIB sizes. CFW and OFW are about 5x slower than BFW in CP construction. Because Cuckoo Filtable is faster to construct than Othello and the two-level design degrade the construction performance of CFW CP. However, the two-level design is necessary to make the data plane memory consumption times smaller than the plain Cuckoo hashing approach, which stores addresses. The high variation of control plane construction time in OFW is because of the varying number of rebuild times. In contrast, the CFW faces much less rebuild during the construction.

Fig. 22 shows the construction time from the CP to a single DP at different FIB sizes. CFW and OFW data plane constructions are fast because of our ‘skeleton’ design. The addresses are MAC addresses. The construction involves value reassignments because CP stores the mapping from addresses to hosts while the FIB in a DP is a mapping from addresses to links. CFW is fast because the value reassignment is simply traversing over slots. In OFW, the value reassignment involves traversing connected components, which exhibits less locality than that of CFW.

C. Evaluation in a real network

We conduct both single-thread and multi-thread forwarding experiments to evaluate the throughput of different forwarders. The multi-thread experiments run on the DPDK built-in poll mode.

We first evaluate the maximum forwarding capacity of Node 1 by an ‘empty’ forwarder that loads the key from each packet and transmits it to Node 2, without looking up any FIB or table. The maximum capacity is 28.40Mpps for 64-byte L2 packets.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

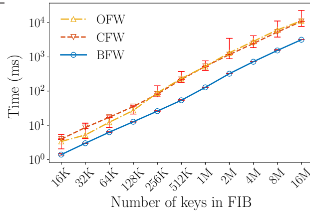


Fig. 21. CP construction time

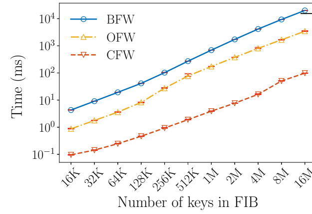


Fig. 22. Single DP construction time

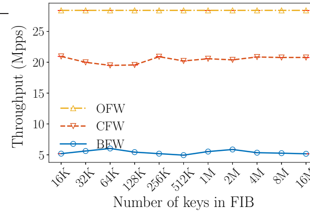


Fig. 23. DP throughput for Zipfian in CloudLab (single thread)

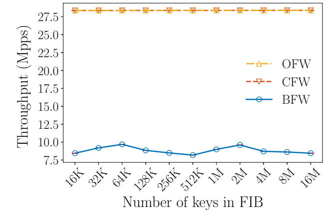


Fig. 24. DP throughput for Zipfian in CloudLab (two threads)

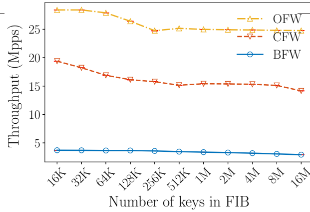


Fig. 25. DP throughput for Uniform in CloudLab (1 thread)

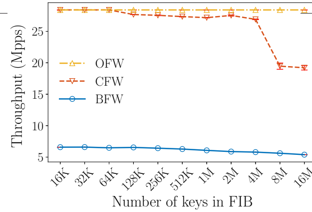


Fig. 26. DP throughput for Uniform in CloudLab (2 threads)

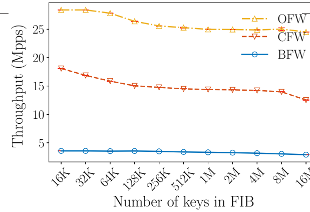


Fig. 27. DP throughput in CloudLab (invalid MACs, 1 thread)

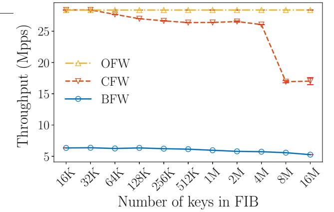


Fig. 28. DP throughput in CloudLab (invalid MACs, 2 threads)

Throughput. Figures 23 to 26 show the throughput of BFW, CFW, and OFW where the forwarding keys are valid MAC addresses. We vary the total amount of addresses stored in the FIB and observe the throughput. The forwarders have lower throughput under uniform key distribution because the memory access pattern exhibits lower locality. OFW performs the best among the three on both single thread and two threads. While single thread OFW almost reaches the forwarding capacity, two threads of OFW are sufficient to reach the forwarding capacity for a 16M FIB. Throughput for Zipfian distribution grows for all three forwarders because their memory access patterns have more locality. CFW on two threads also reaches the forwarding capacity. For all cases, OFW and CFW perform >2x better than BFW.

D. Summary of comparison.

Throughput. OFW and OGW exhibits >2 times lookup throughput compared to CFW and CGW. In other cases, the throughput of OFW and CFW are similar. The lookup throughput of BFW is $< 10\%$ compared to the other two.

Memory footprint. (Evaluated and compared in Section V-D) When alien addresses are not a concern, such as in core switches, OFW costs the least memory. The memory cost of CFW and BFW are similar. When we need to filter alien addresses, such as on gateway switches, the memory cost of OFW is higher than that of CFW or BFW.

Incremental update. OFW and CFW can perform > 10M updates per second, while BFW is much slower than them.

VIII. Insights and Discussion

Design consideration by network operators. For networks using name-based forwarding, there are two types of forwarding nodes: gateway nodes and core nodes. On gateway nodes, CFW provides the lowest false positives rates given the same memory budget. *Hence, CFW and potentially other Cuckoo variants in future are ideal design choices for gateway nodes.* On core nodes, false positives are not a consideration. OFW provides the highest throughput and lowest memory cost compared to other solutions. *Hence, OFW and potentially other Othello variants are ideal design choice for core switches/routers.* In all situations studied in this paper, *BFW, the Bloom filter based solution, is not the best choice*.

Further optimization. From the results, the performance of Cuckoo Filtable downgrades dramatically compared to Cuckoo hashing. Design optimizations are possible but hard. It is difficult for a Cuckoo hashing based FIB to store a small number of addresses to achieve memory efficiency while avoiding valid key collisions, which lead to key shadowing described in § IV. The implementation of collision avoidance sets at Level 1 of CFW FIB can be further improved because we store full keys in the sets instead of memory addresses of the keys, which may waste memory and in turn downgrade the construction performance. An adaptive Cuckoo filter (ACF) [33] is a filter for approximate membership queries, rather than a key-value lookup table that can be used for forwarding. It costs more space to resolve false positives, and it cannot avoid valid key collisions which lead to key shadowing.

IX. Conclusion

This work provides a comprehensive study of redesigning DCSes for packet forwarding with network names in multiple network models. By utilizing the programmable network model, we propose new forwarding structure designs based on three representative DCSes: BFW (based on Bloom filter), CFW (based on Cuckoo hashing), and OFW (based on Othello hashing). They improve existing non-programmable-network methods by **a big margin** in both memory efficiency and control plane scalability. The analytical and experimental comparison among these three methods reveals that CFW and OFW fit various network setups that can be chosen by network operators, while BFW may not be ideal in most cases.

X. Acknowledgement

This work is partially supported by National Science Foundation Grants 1717948 and 1750704. We thank the shepherd Fernando Kuipers and the anonymous reviewers for their suggestions and comments.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.

References

[1] CloudLab. https://www.cloudlab.us/.

[2] Implementation of farmhash. https://github.com/google/farmhash.

[3] Implementation of Othello: a concise and fast data structure for classification. https://github.com/sdyy1990/Othello.

[4] Intel DPDK: Data Plane Development Kit. https://www.dpdk.org.

[5] Pktgen-DPDK. https://github.com/pktgen/Pktgen-DPDK.

[6] Technical report: detailed derivations of performance metrics. . https://mybinder.org/v2/gh/sshi27/Re-designing-Compact-structure-based-Forwarding-for-Programmable-Networks/master?filepath=design.ipynb.

[7] Network Functions Virtualisation: Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[8] Implementation of presized cuckoo map. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/presized_cuckoo_map.h, 2016.

[9] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. In *Proc. of ACM SIGCOMM*, 2010.

[10] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. of ACM SIGCOMM*, 2008.

[11] D. Belazzougui and F. C. Botelho. Hash, displace, and compress. In *Proc. of Algorithms-ESA*, 2009.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[14] A. T. Campbell, H. G. D. Meer, M. E. Kounavis, K. M. Hitachi, J. B. Vicente, and D. Villela. A survey of programmable networks. *SIGCOMM Computer Communication Review*, 1999.

[15] D. Charles and K. Chellapilla. Bloomier Filters: A Second Look. In *Proc. of European Symposium on Algorithms*, 2008.

[16] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA*, pages 30–39, 2004.

[17] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*, 2016.

[18] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014.

[19] B. Fan, D. Zhou, H. Lim, M. Kaminsky, and D. G. Andersen. When cycles are cheap, some tables can be huge. In *Proc. of USENIX HotOS*, 2013.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 2000.

[21] N. Feamster, J. Rexford, and E. Zegura. The road to SDN: An intellectual history of programmable networks. *ACM Queue*, 2013.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, 2009.

[23] A. Hari, T. V. Lakshman, and G. Wilfong. Path Switching: Reduced-State Flow Handling in SDN Using Path Information. In *Proc. of ACM CoNEXT*, 2015.

[24] C.-Y. Hong et al. Achieving High Utilization with Software-Driven WAN. In *Proceedings of ACM Sigcomm*, 2013.

[25] S. Jain, Y. Chen, S. Jain, and Z.-L. Zhang. VIRO: A Scalable, Robust and Name-space Independent Virtual Id ROuting for Future Networks. In *Proc. of IEEE INFOCOM*, 2011.

[26] S. Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proceedings of ACM Sigcomm*, 2013.

[27] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. of Sigcomm*, 2008.

[28] E. Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, 2000.

[29] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of ACM SOSP*, 2011.

[30] B. M. Maggs and R. K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 2015.

[31] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. A Family of Perfect Hashing Methods. *The Computer Journal*, 1996.

[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[33] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

[34] M. Moradi, F. Qian, Q. Xu, Z. M. Mao, D. Bethea, and M. K. Reiter. Caesar: High-Speed and Memory-Efficient Forwarding Engine for Future Internet Architecture. In *Proceedings of ACM/IEEE ANCS*, 2015.

[35] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks . *IEEE Communications Surveys and Tutorials*, 2014.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

[37] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[38] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*, 2015.

[39] C. Qian and S. Lam. ROME: Routing On Metropolitan-scale Ethernet . In *Proceedings of IEEE ICNP*, 2012.

[40] C. Qian and S. Lam. A Scalable and Resilient Layer-2 Network with Ethernet Compatibility. *IEEE/ACM Transactions on Networking*, 2016.

[41] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani. MobilityFirst: A Robust and Trustworthy MobilityCentric Architecture for the Future Internet. *Mobile Computer Communication Review*, 2012.

[42] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, 2001.

[43] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. Mckeown, and J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proc. of the ACM SIGCOMM*, 2016.

[44] S. Shi, C. Qian, Y. Yu, X. Li, Y. Zhang, and X. Li. Concury: A Fast and Light-weighted Software Load Balancer. *arXiv:1908.01889*, 2019.

[45] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5), 2016.

[46] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of ACM SIGCOMM*, 2002.

[47] M. Wang, C. Qian, X. Li, and S. Shi. Collaborative Validation of Public-Key Certificates for IoT by Distributed Caching. In *Proc. of IEEE INFOCOM*, 2019.

[48] T. Yang et al. A shifting bloom filter framework for set queries. In *Proceedings of VLDB*, 2016.

[49] T. Yang et al. Coloring embedder: a memory efficient data structure for answering multi-set query. In *Proceedings of IEEE ICDE*, 2019.

[50] K.-K. Yap et al. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proc. of ACM SIGCOMM*, 2017.

[51] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. of ACM CoNEXT*, 2009.

[52] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang. A concise forwarding information base for scalable and fast name lookups. In *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*, 2017.

[53] Y. Yu, X. Li, and C. Qian. 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)*, 2017.

[54] Y. Yu and C. Qian. Space shuffle: A scalable, flexible, and high-bandwidth data center network. In *Proceedings of IEEE ICNP*, 2014.

[55] D. Zhou, B. Fan, H. Lim, D. G. Andersen, M. Kaminsky, M. Mitzenmacher, R. Wang, and A. Singh. Scaling up clustered network appliances with scalebricks. In *SIGCOMM*, 2015.

[56] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of ACM CoNEXT*, 2013.

Authorized licensed use limited to: Univ of Calif Santa Cruz. Downloaded on October 15,2021 at 21:42:45 UTC from IEEE Xplore. Restrictions apply.