UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ALGORITHMIC AND SYSTEM INNOVATIONS FOR NETWORK DATA PLANE: EFFICIENCY, SCALABILITY, AND FLEXIBILITY

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Shouqian Shi

March 2021

The Dissertation of Shouqian Shi
is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Algorithmic and System Innovations for Network Data Plane: Efficiency, Scalability, and Flexibility

by

Shouqian Shi

Due to the advanced reliability, scalability, and cost-effectiveness, more and more businesses are turning to cloud computing. Large-scale cloud networks have been connecting users, data, and machines more tightly than any past time. According to Forbes, cloud computing is enjoying a more than 15 percent growth per year in the global market size. And, Flexera reports that more than half of the surveyed companies, being enterprise or small businesses, are using more cloud services than they expect due to the impact of COVID-19. Among the surveyed companies, the top concern in cloud computing is cost-effectiveness. However, Moore's law fails in recent years because the cost for a single gate of an integrated circuit is not decreasing anymore. Hence, architectural reorganizations and algorithmic innovations are two main approaches to achieve higher effectiveness in the post-Moore's law era.

Most cloud networks require high capacity Forwarding Information Bases (FIBs) to support massive network traffic from numerous end devices. The growth of the FIB limits the performance of network operations and increases infrastructure costs. We propose to reorganize the standard SDN model's functions and extract the common

update calculations from the data plane to the control plane [**ICNP'19**]. We call this 'skeleton-based update'.

This dissertation presents a new algorithm, Ludo hashing [**ACM SIGMETRICS'20**] for fast key-value lookup, based on the reorganized skeleton-based update model for SDN. Ludo achieves the most compact memory cost among all alternative algorithms by saving 40% to 80%+ space than existing dynamic solutions. Ludo hashing is specially designed for cloud computing and distributed systems and is ready to be applied to many applications, *e.g.* , network forwarders, Content distribution network (CDN), cloud load balancers, Network Address Translation (NAT), and data sharing or collaboration tasks for IoT devices. We then designed Concury [**SOCC'20**], a fast and light-weight software load balancer for cloud networks. Concury improves the throughput by >2x and costs the smallest memory compared to state-of-the-art L4LB algorithms while providing weighted load balancing. Concury is read-only during connection establishments and terminations, while the connection consistency is still guaranteed by design.

To my wife,

Lily Li,

you are the song in my life.

Chapter 1

Introduction

Fast lookups of large-scale key-value items are fundamental functions and design blocks of numerous networked and distributed systems. These *in-memory key-value lookup engines* serve as the indices to store and find the locations, addresses, or directions of the destination devices or queried data. The representative applications of these lookup engines include:

1. The forwarding information bases (FIBs) on network routers and switches run in SRAM. Many FIBs use key-value lookup engines to forward packets in data center networks [56, 53, 50, 104], metropolitan networks [80], LTE [110], software defined networks (SDNs) [111, 107], and future internet designs [83], by searching flat network addresses, as such MAC. The values of the lookup are outgoing packet ports.

2. In a content distribution network (CDN) or edge network, a number of proxy servers cache popular internet contents [65, 102, 85]. A lookup table can be used

to find the server that stores a particular content [43].

3. In a distributed file system, an index is required to maintain metadata and the location of file storage [76, 97]. The lookup keys are usually file names or IDs, and the values are locations where the files are stored.

4. Cloud load balancers are important components of a data center, which distribute packets to replicated backend servers [77, 37, 67]. Here the lookup engine stores the flow states where each key is a 5-tuple, and each value is a server index. Network address translation (NAT) also stores flow states and performs lookups based on 4-tuple for every packet.

5. In embedded IoT devices, lookup tables are required for sharing sensing data and public keys [60, 93].

This dissertation identifies three important and connected problems in improving the efficiency, scalability, and flexibility of key-value lookup algorithms with applications in network data plane functions. We propose the solutions with algorithmic innovations to address these problems and implement the solutions in real prototype systems.

1.1 Architectural reorganization for higher update efficiency in SDN networks.

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 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 attention 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 [50][109], enterprise networks[104][111], and metro-scale Ethernet [56][53][80][81], where the key is the MAC or other L2 addresses. Although many existing data centers employ the fat-tree based design that uses IP routing, name-based routing still provides a number of advantages, including flexible management and host mobility. L2 name-based architectures are also suggested in many future network proposals [56][53][80].

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 [111][51][78][86]. The matching key is some header fields. In addition, many new internet architectures suggest flat-name forwarding in the network layer, such as MPLS [84], LTE [110], Mobility-first [83], and AIP [24].

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 [43][65]. The emerging edge computing provides more sophisticated content/service caching services [89][102].

Unlike IP addresses, aggregating network names is challenging – 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. 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 [43][104][71], Cuckoo hashing [111][110], and Bloomier filters [35][33][105][108], to resolve this problem. 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 costs by using a small amount of memory.

2. **Fast lookups.** Faster lookups mean higher forwarding throughput. The through-

put 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.

Although many FIB algorithms have been proposed, the recently developed programmable network paradigm [30][32], such as Software Defined Networks [44][72] and Network Functions Virtualization [9], still provides the potential to further reduce the time and memory complexity of forwarding algorithms. Hence, there is a need for re-designing forwarding algorithms with the DCSes under this new paradigm. To our knowledge, **there is no existing work that fully explores the potential of programmable networks for FIB designs**. The only exception is the very recently proposed Othello hashing [105] based on Bloomier filters, but it has no network-wide design, as explained later. We design and implement **new forwarding algorithms** for programmable networks by **re-visiting** three representative DCSes: Bloom Filters [28], Cuckoo hashing [74], and Bloomier filters (Othello hashing) [35][33][105].

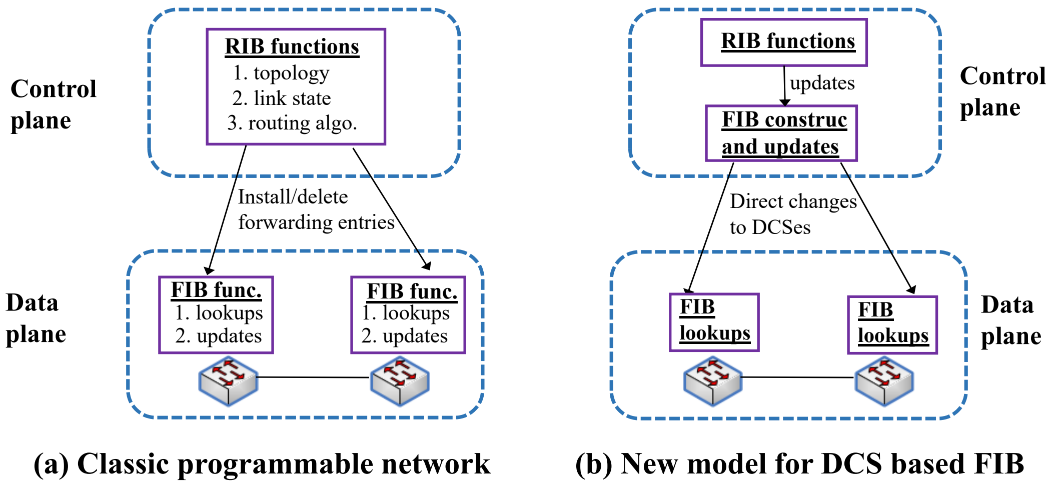


Figure 1.1: Separating a DCS-based FIB into two planes

As shown in Fig. 1.1(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 shown in Fig. 1.1(b). We relocate the memory and computation of the update function from many FIBs to the central control plane while the data plane FIBs, supporting direct updates, focus on fast lookups. Our approach significantly reduces the data plane memory footprint while preserving 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 into 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.

1.2 Ludo hashing: algorithmic innovation for higher lookup efficiency in key-value lookup systems.

The important requirement of in-memory lookup engines (such as those for name based forwarding) is **space efficiency**. It is because they are hosted in high levels of the memory hierarchy or special network devices, where the memory is fast, small, expensive, and power-hungry. Another requirement is to support **dynamic updates** that allow the tables to work in practice, including key-value insertions, deletions, and changes.

Hash tables are the conventional solutions of fast in-memory key-value lookup. To resolve hash collisions, the item keys should be stored to tell which value belongs to which key. For example, the widely used version of Cuckoo Hashing [74] allows up to 8 key collisions [38, 40, 62, 111, 110]. Hence Cuckoo Hashing must store the keys or at least the digests of keys [64]. The digest of a key is the hash value of the key, usually truncated to suit the application. For example, a Cuckoo filter may use 8-bit digest for a false-positive rate at 3%. *Storing keys may cost more space than storing the values* in the above applications. For example, a typical file ID in a storage system has hundreds of bits, and each value (disk address) is only tens of bits. For FIBs, the network addresses (48 to > 100 bits) are longer than the port values ($\leqslant 8$ bits). In a CDN, the keys (URLs) could be thousands of bits.

Hence, recent efforts have been made to use *minimal perfect hash functions (MPHFs)* [27, 39, 49] for in-memory key-value lookups, which significantly reduce the space costs by avoiding storing keys. For a set of n key-value items where each item is a tuple (k_i, v_i) of key k_i and value v_i, a minimal perfect hash function H' maps the n keys to integers 0 to $n-1$ *without collision.* The lookup table can simply use the MPHF and an array of n values, where the i-th value corresponds to the key that is mapped to i by H'. The lookup table does not need to store keys. Unfortunately, none of the existing MPHFs support fast dynamic updates. When there is a single item insertion/deletion, the MPHF and whole array that stores values need reconstruction. Bloomier filters [35, 33] and SetSep [42, 110] are two alternative perfect hash tables that have been used for network applications [107, 101, 108, 93, 87, 42, 110]. However, Bloomier filters introduce 100% space overheads to store the values, and SetSep is difficult to update, as explained in § 4.2.

This work presents Ludo hashing, a space-efficient lookup engine based on perfect hashing, which supports $O(1)$ lookups and dynamic updates. To our knowledge, Ludo hashing *induces the least space overheads* compared with existing solutions of dynamic key-value lookups. We show the numerical comparison of these solutions in

Table 1.1 and Fig. 1.2.

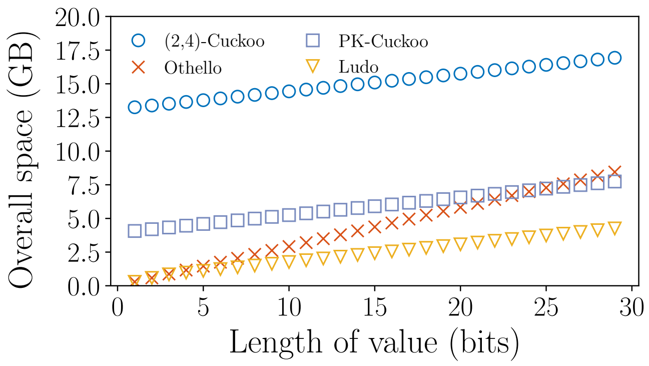


Figure 1.2: Numerical space comparison of dynamic key-value lookups. $n = 1$ billion, $L = 100$ bits, $L' = 30$ bits. Ludo uses Bloomier for $l < 4$. n: number of items in table.

Ludo hashing enables space savings by removing the key storage while maintaining a low amplification factor (AF) on values. AF is the number of additional bits taken per item when the length of values is incremented by 1 bit. For example, SetSep takes $0.5 + 1.5l$ bits for l-bit values, and the AF of SetSep is 1.5. The *core idea* of Ludo hashing can be presented in two steps. **Step i)**: We first insert all key-value items into a Cuckoo hash table. In this way, the key-value items are divided into a number of small groups, where each group only contains at most four items. **Step ii)**: For each group, we find a hash function H such that H maps the four keys to integers 0 to 3 *without collision*. For most modern random hash function algorithms, we may generate an independent hash function H_s by using a different seed s. Hence we find the right hash function for each group by trying different seeds using *brute-force*. Within each group, it is only necessary to store one seed s and four values that are in the order of the result of $H_s(k)$ for each key k. Both steps cost $O(1)$ time during lookups, and each insertion/deletion/change can be updated in $O(1)$ amortized time. Eventually, we save the space of storing four keys — hundreds of bits or more — by using a 5-bit seed, found within 31 tries.

The main contribution of this work is a dynamic key-value lookup engine that achieves the least memory cost among existing methods to our knowledge and supports fast post-construction updates. It is based on our discovery of a minimal perfect hashing method with an $O(1)$ update cost. The compactness in a dynamic system is achieved via a novel combination of Bloomier filters, Cuckoo hashing, and brute-force based slot arrangement. We have implemented the complete software of Ludo hashing with dynamic updates and single writer/multiple readers concurrency. We implement and evaluate Ludo hashing in two working systems deployed in a real cloud environment. Experimental and analytical results are available for each design choice to inspire future methods and tools. The **source code** of Ludo hashing is available for result reproducibility [3].

1.3 Application of the SDN reorganization and Ludo hashing for cloud load balancers.

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 [18, 16, 13] have limitations on scalability, availability, flexibility, and cost-efficiency [37]. Hence, major web services such as Google [37], Microsoft [77], and Facebook [10, 36] have started to rely on 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*. For example, the first packet of a TCP connection is stateless, and the following packets of the same connection are stateful. 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* to randomly pick a backend server to serve it. The weight is 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* where all packets of the same TCP connection are served by the same backend server to preserve the connection nature of TCP on each backend server.

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 in the data plane [77, 37]. These stateful LBs experience a 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 [47] and Google [37]. 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 [37]. This design has two major weaknesses: 1) using long digests may still require a large amount of memory while using short digests causes violations 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 [67] – 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 [40, 63] cannot be easily applied to LB algorithms because they only work with full keys rather than digests, introducing prohibitively high memory overhead in large-scale networks. Recent work uses ASICs on programmable switches for fast table lookups [67] while increasing 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

[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.

succinct in memory footprint, 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 work only focuses on LBs.

Compared to existing stateful LBs [77, 47, 37, 10, 67], Concury provides two main advantages. 1) We realize that the current limitations of software LBs stem 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) [37, 67]. The drawbacks include 1) false table hits due to digest collisions [67] and 2) table explosion due to the difficulty of removing digests. Concury uses a data structure to represent all packet states in a succinct manner (just two small arrays, $2.33n$ elements in total, where n is the number of keys) by utilizing the theoretical foundation of minimal perfect hashing [26, 66, 35, 34, 106]. 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, no false hits, and data plane updates 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 on 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 [89, 103, 85, 102]. 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 [89, 103, 108]. 2) Traditional cloud LBs consider a state for every TCP connection. In modern cloud or edge, a single device may host multiple virtual machines, and hence the states may be for *multiple destinations* at both the device level and the process level [89, 103, 108]. 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 [89]. This whole process consists of multiple TCP flows and UDP pseudo-flows, all of which should be sent to a consistent DIP. Unlike previous designs, Concury naturally support multi-connection states. 3) Modern cloud and edge servers might have heterogeneous capacities in computation, storage, and bandwidth [89, 103]. Different capacities appears as different weight in load balancing. 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 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 [5] to shows its high performance **in two real networks**. We also build a P4 prototype to show its compatibility with programmable switches. The source code can be accessed here [22], and our results can be reproduced.

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 zero false hits, 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 also supports dynamic state maintenance that is useful for other network functions and future applications.

Solution	Space cost (bits per item)	Lookup time per query	Update time per operation
MPHF +Array	$> 1.44 + l$ (*)	$O(1)$, >67ns [39]	Not allowed
SetSep [42, 110]	$0.5 + 1.5l$	$O(1)$, 212ns	>120**ms**
Partial key Cuckoo [64]	$1.05(L' + l)$	$O(1)$, 163ns	>46ns [87]
Bloomier/Othello [35, 33, 107]	$2.33l$	$O(1)$, 187ns	173ns
Ludo hashing (this work)	$3.76 + 1.05l$	$O(1)$, 303ns	163ns

Table 1.1: l: bit length of each value. L: bit length of each key. L': bit length of each key digest. The empirical values are from our experiments with 64M keys and $l = 20$ as explained in § 4.6. (∗)The most compact version of MPHF [39] costs $1.56 + l$ bits per item, already at a prohibitively high construction time cost: 2ms per item. The SetSep papers [42, 110] include neither clear update function nor experimental results of updating. We designed an update function in our best effort.

Chapter 2

Background Algorithms

In-memory key-value lookup algorithms with small memory footprint support vital functions of many networked and distributed systems, including network forwarding [104, 111, 107, 110], distributed storage [76, 97], cloud load balancers [67], and content distributions [43, 65]. Our contributions are related to the following key-value lookup algorithms.

2.1 Bloom filters

The Bloom filter [28] is one of the most popular *dynamic and compact data structures* (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. As shown in Fig. 2.1, a Bloom filter is a bitmap associated with n_h hash functions, *e.g.*, $n_h = 3$ in this example. To construct a Bloom filter, each element in the key set K is inserted sequentially. For each key k, n_h hash values of k are calculated

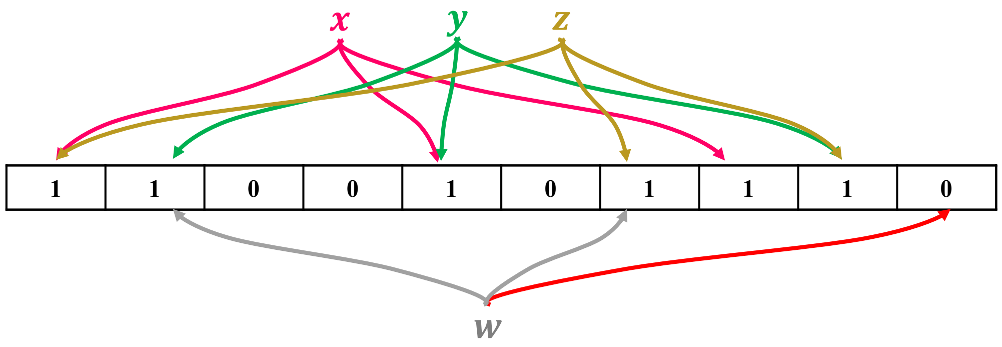


Figure 2.1: Bloom filter for set $\{x, y, z\}$

via the n_h hash functions. The n_h hash values index n_h bits in the bitmap, and these bits are set to 1. To lookup a key k, the n_h hash values are calculated, and the indexed n_h bits in the bitmap are checked to see if all bits are 1. In this example, the key w fails the check. A well-known feature of Bloom filters is that its results include false positives but no false negatives.

2.2 Cuckoo hashing

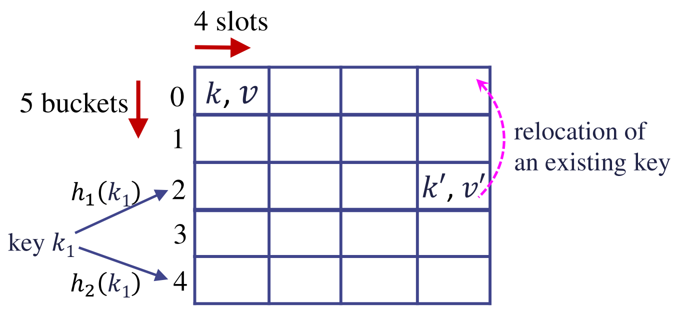


Figure 2.2: (2,4)-Cuckoo Hash Table

Cuckoo hashing [74] is a key-value mapping data structure that achieves $O(1)$ lookup time in the worst case and $O(1)$ update time on average. As shown in Fig. 2.2:

a (2,4)-Cuckoo has a number of buckets, each bucket has 4 slots, and every key-value pair is stored in one slot of the two *alternate buckets* based on the two hash values $h_0(k)$ and $h_1(k)$. The lookup of the value for a key k is to fetch the two buckets and match the keys in all 8 slots until a key matches k correctly. For an item insertion with key k_1, a single empty slot should be found in bucket $h_0(k_1)$ or $h_1(k_1)$. If both the buckets are full, one existing item (e.g., the one with key k' in Fig. 2.2) will be relocated to the other alternate bucket of k', and k_1 takes the slot of k'. If the alternate bucket of k' is full as well, an item in that bucket will be relocated recursively. This process stops when every item is placed in a slot. Many recent system designs choose the (2,4)-Cuckoo hashing [41][111][110] to maximize the memory load factor.

2.3 Bloomier filters

We propose to use the data structures and algorithms of minimal perfect hashing [26, 66, 35, 34, 106] for many key-value lookup systems, including network forwarders, cloud load balancers [67], and content distributions [43, 65]. One well-known family of perfect hashing based data structures are Bloomier filters [35, 34]. The recently proposed Othello Hashing [106, 107] 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, Oth-

k	value	$h_a(k)$	$h_b(k)$
k_1	01	6	5
k_2	10	1	0
k_3	11	1	2
k_4	00	1	3
k_5	10	4	2

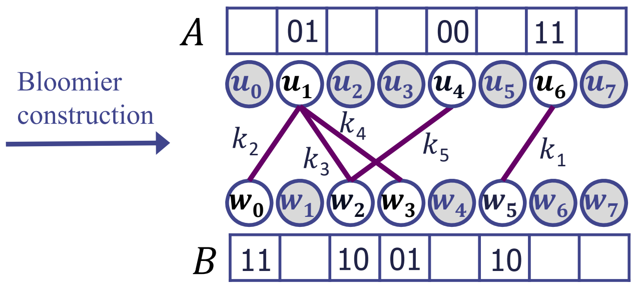
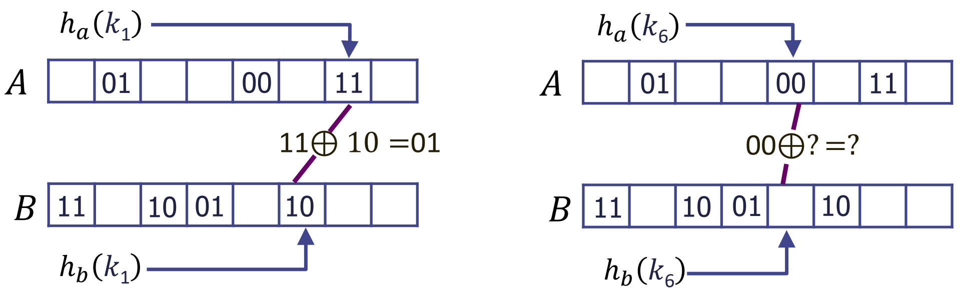

Bloomier construction
A
01
00
11
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7
k_2
k_3
k_4
k_5
k_1
w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7
B
11
10
01
10

Figure 2.3: Construction of Othello hashing

ello 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 point is detailed in § 5.4.

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 value mapped to the key.

Bloomier filter construction in the Othello control plane. We use an example in Fig. 2.3 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



(a) Look up a key known during construction: Specified Result **(b) Look up a key unknown during construction: Deterministic Random**

Figure 2.4: Lookups of Bloomier filter

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*. If G is acyclic, it is trivial to find a valid element assignment such that the values of all keys are satisfied [106]. If a cycle is found, the construction needs to find another pair of hash functions to rebuild G. It is proved that during the construction of n keys, the expected total number of re-hashings is < 1.51 when $n \leqslant 0.75m$ [106]. 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. 2.4 (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. 2.4(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 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 § 5.4.4.

Chapter 3

Re-designing Compact-structure based Forwarding for Programmable Networks

3.1 Overview

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. 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. We realize that there is a lack of comprehensive study and comparison of these methods in different network layers and situations. In addition, recently developed programmable networks have the potential to further reduce the time/memory complexity of forwarding structures with new designs.

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 study two representative cases of name-based forwarding, namely L7 overlay content lookup and L2 address-based forwarding. 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 into 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 work.

3.2 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 construction and lookup of the Bloom filter [28] is previously introduced in § 2. 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 names/addresses that should be forwarded to this link, such as Summary Cache [43] and BUFFALO [104]. Each lookup takes $O(n_h \cdot d)$ time, and each update takes $O(d + n_h)$

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 [99] achieves fast lookups, but its false positive rate is high.

Cuckoo hashing. The construction and lookup of the Cuckoo hashing [74] is introduced in § 2. FIBs using Cuckoo hashing store the link or port index in each 'value' field together with the key (name), such as CuckooSwitch [111]. ScaleBricks [110] uses both Cuckoo hashing and SetSep [42] 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 [35][33] and their variants Othello hashing [105][94] and Coloring Embedder [100] are key-value lookup tables inspired by dynamic perfect hashing [66][27]. The construction and lookup of the Othello hashing is introduced in § 2. The important features of Othello are 1) the memory cost is small as it stores no keys in the lookup 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 [105] 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. 2.4(b). SDLB [108] and Concury [88] are L4 load balancers using Othello.

3.3 Network models

We study the network forwarding problems of L2 and L7 networks with modern programmable networks. We give a general abstraction of both network architectures and formalize the corresponding forwarding problems.

3.3.1 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, firewalling, load balancing, and traffic monitoring. The typical examples include SDNs [69][54][52][44][72][30], software routing and switching [59][78][111][51][86], and network functions virtualization (NFV) [9][75][37]. We observe that the programmable network paradigm provides *a new opportunity* to allow new data structures and algorithms to run on network devices and their further optimizations. As shown in Fig. 1.1(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 a 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.

3.3.2 Layer-2 Forwarding

L2 applications. In a modern data center network, there are a massive amount of physical servers [56][50][80]. Each server has an ID (e.g., its MAC address). An interconnection of switches connects the servers. Each switch has multiple ports connecting neighboring switches and servers. A switch in a data center may be a **gateway** that connects to external networks 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 server. A switch forwards the packet to a neighbor based on FIB lookups using the MAC. Many modern networks are variants of this model [56][50][80]. For example, in a multi-tenant data center network, multiple VMs are hosted in one physical server. The FIB serves the same function with the only difference: multiple VMs may connect to a switch through a single port. This model is not limited to L2. If MAC is replaced with other network IDs or addresses in other layers, the FIB still provides similar functions. For flow-based networks [69], the flow ID may be a combination of source/destination IPs, MACs, and other header fields. The forwarding may be on a per-flow basis rather than a 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 [110].

L2 networking model. There are a total number of n switches connected with each other to form an L2 network. Each port of a switch is linked to either a server or another switch. The maximum number of ports is fixed as d. Each server has an address k as its unique ID or search key. The forwarding structure on each switch should include the server-port mapping of *all* servers in the network. The port information of k indicates the next-hop node to route the packet to the server k. A switch may receive network packets sent to any server.

In programming networks, the switch fabric only includes the DP. And there is a logically centralized CP, possibly running on a separate server [72]. The CP executes the routing and other protocols and sends updates to the switch DP. We do not compare routing protocols in this work and focus on forwarding.

3.3.3 Layer-7 Forwarding

L7 applications. Many L7 overlay networks conform to the following model. For an L7 CDN such as Akamai, a large number of data contents provided by the data center of the service provider are cached among the CDN hosts (i.e., cache servers) across diverse geographic regions [65]. The content cache can help to significantly reduce the latency of content downloading requested from the users. When a user is requesting a content by its URL, the DNS server will direct the user request to the nearest CDN host. Due to the massive volume of content resources, a single CDN host cannot hold all of them. Hence, a CDN host may forward the request to another nearby CDN host that caches the requested content. Hence, each CDN host should maintain a forwarding table, including the ID/URL of all contents and the corresponding neighboring hosts. A CDN may also use consistent hashing [55][65] for content indexing. The major weakness of consistent hashing is that it cannot preserve content locality: a content will be placed to an arbitrary host based on hash values, possibly far from its original users. We only consider forwarding table based CDN content indexing [43]. This model may be applied to other networks such as distributed data storage [23], P2P systems, or edge computing [89].

L7 networking model. There are a total number of n nodes (e.g., CDN hosts) in the overlay network that stores contents. Each content has a unique ID k that could be a URL. The set of all cached contents K in a region has the cardinality n_k. Each node has a certain number of neighbors and maintains an index of a subset of K. These subsets may arbitrarily overlap. Every node has a local content list to remember the IDs of contents stored on itself. A FIB is maintained at every node which stores the content-location mapping, where the location could be the IP address of the node that stores the content. Upon a content lookup, the ID of the node holding this content is returned. If no node stores the content, *null* is returned, and the request should be forwarded to the remote data center. The contents are stored into and removed from the nodes frequently, so the design of the FIB should also take dynamics into consideration.

Each node runs the FIB (DP component) locally. There is a separate program running the CP component, which could be either local or remote. In this work, we do not compare cache replacement algorithms – they are out of scope.

3.3.4 Comparing L2 and L7

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. Routing in L2 will usually take multiple hops from the source to the destination. L7 routing paths are much shorter in this model. 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 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).

3.4 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 [28], Cuckoo Forwarder (CFW) based on a new data structure Cuckoo Filtable, and Othello Forwarder (OFW) that extends Othello hashing [105]. **CFW and OFW are considered our main design contributions, and BFW is a baseline for comparison.**

3.4.1 Bloom Forwarder (BFW)

Limitations of existing methods. Both BUFFALO [104] and Summary Cache [43] 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 [43] 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 [104] 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 all original keys to build new BFs.

Bloom Forwarder (BFW). BFW uses a similar DP design to BUFFALO [104], but a different CP design. We follow the extensive optimizations proposed in

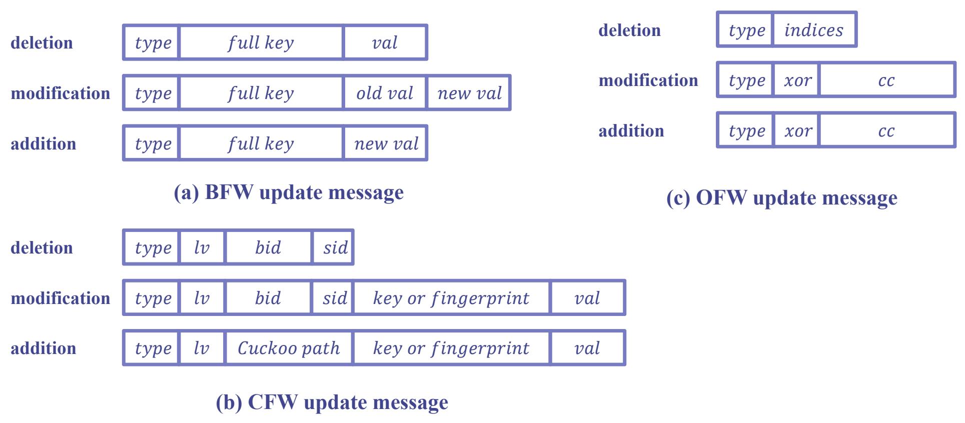
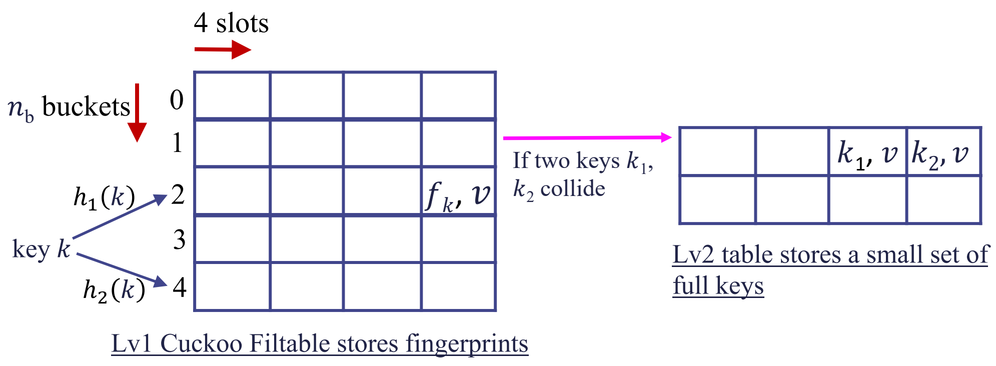


Figure 3.1: Update messages for forwarders

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.

One possible question is whether the DP could maintain $\log_2 d$ BFs of the same size rather than d BFs, such that the replies from all $\log_2 d$ BFs can form a $\log_2 d$-bit long index that represents a set of keys that should be forwarded to each link. After careful study, we find that this method is very memory-inefficient and may cause high false positives, mainly due to the different cardinalities of the key sets of these BFs. We explain this reason in details in Appendix A.1.1.



4 slots
n_b buckets
0
1
2
3
4
$h_1(k)$
key k
$h_2(k)$
f_k, v
If two keys k_1, k_2 collide
k_1, v
k_2, v
Lv1 Cuckoo Filtable stores fingerprints
Lv2 table stores a small set of full keys

Figure 3.2: Example of Cuckoo Filtable

3.4.2 Cuckoo Forwarder (CFW)

Limitations of existing methods. CuckooSwitch [111], 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 [41] 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. 3.2. 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

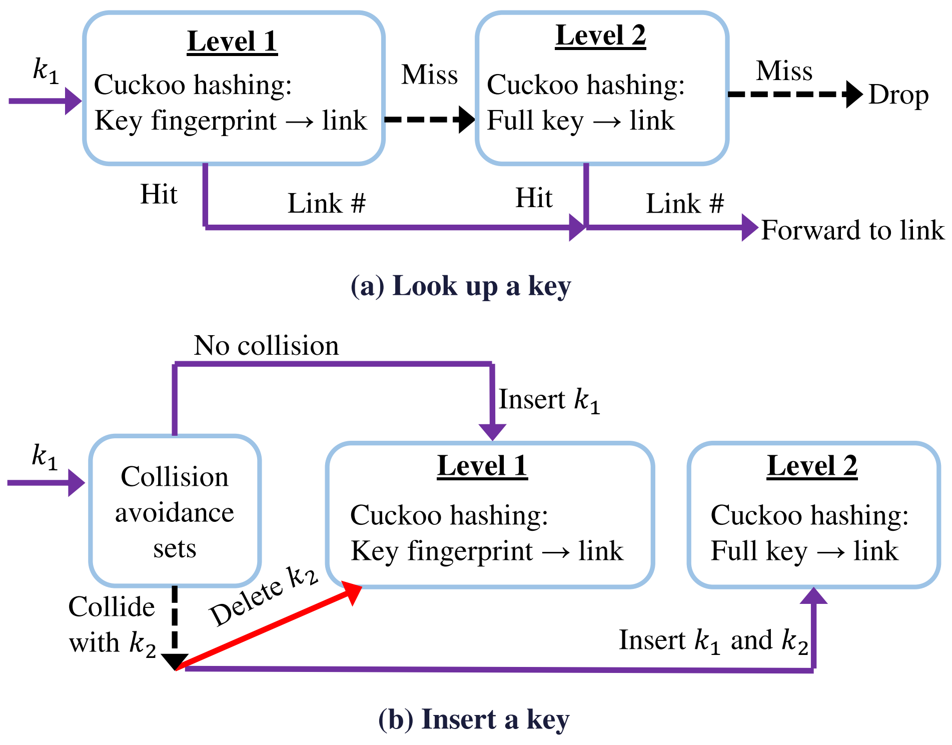


Figure 3.3: CFW insertion and lookup workflows

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 [64], the Cuckoo Filtable addresses the following challenges.

Challenges in CFW DP design. By storing the fingerprints, CFW experiences *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 [56], 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 *key shadowing*.

To resolve valid key collisions, we adopt a two-level design, as shown in Fig. 3.2. 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 into 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 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. 3.3(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 with 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. It means that the CFW CPs on different nodes will have the same size of hash tables, and the slots of the same location store the same key. The only difference is that their value fields are different in L2 – for L7 even the values are the same. **This property has not been 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. 3.3(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.

CFW CP-DP synchronization. The CP to DP synchronization is achieved by incremental updates. The format of a single update message is shown in Fig. 3.1 (b). The two bits for the operation type ranged among 'addition', 'deletion', and 'modification'. The field *lv* is the indicator of the level this update is going to be made in. *bid* and *sid* are common to specify the bucket index and the slot index, respectively. The 'Cuckoo update path' should be sent in a key addition message. Upon receiving an incremental update message, the DP updates the FIB accordingly.

There is a design choice for CFW: align the buckets to the cache line size to achieve fast lookup performance (*aligned*) or place the buckets next to each other to get a smaller memory footprint (*compact*). We choose *compact* as analyzed in [7] and Appendix A.1.4.

3.4.3 Othello Forwarder (OFW)

We further explore the efficiency of Othello hashing [105] for a new FIB design in programmable networks.

Limitations of existing methods. Concise [105] 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

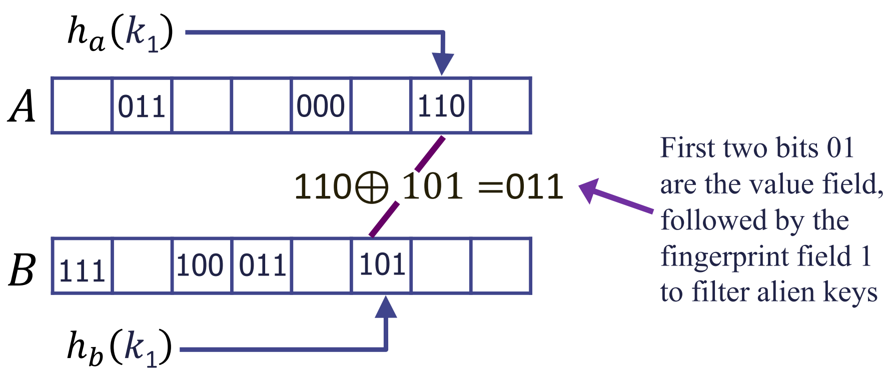
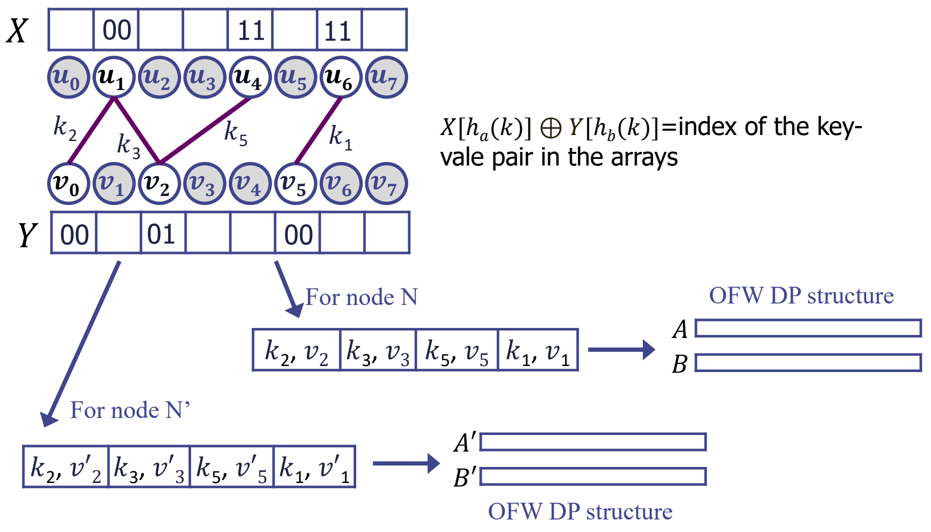

$h_a(k_1)$
A
011
000
110
$110 \oplus 101 = 011$
First two bits 01 are the value field, followed by the fingerprint field 1 to filter alien keys
B
111
100
011
101
$h_b(k_1)$

Figure 3.4: Lookup of Othello hashing with fingerprint

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. 3.4. 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, the 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



X
00
11
11
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7
k_2
k_3
k_5
k_1
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
Y
00
01
00
$X[h_a(k)] \oplus Y[h_b(k)]$=index of the key-vale pair in the arrays
For node N
k_2, v_2 k_3, v_3 k_5, v_5 k_1, v_1
OFW DP structure
A
B
For node N'
k_2, v'_2 k_3, v'_3 k_5, v'_5 k_1, v'_1
A'
B'
OFW DP structure

Figure 3.5: OthelloSet for network-wide updates

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. We prove that this method can detect an alien key k with the probability 63.2% $>$ 50%, as derived in Appendix A.1.2.

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. 3.5, 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 the 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 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, and the CP only has to determine the values to fill the slots in arrays A and B.

3.4.4 Control plane reusing and scalability

The key reason for 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 problems. In fact, the server used in our experiments cannot afford to store CBFs for over 100 nodes. On the other hand, the DPs of CFW and OFW can be reused if different nodes are forwarding the same set of addresses, which is true in many L2 networks [50][56][53][80]. 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 the same subnet.

OFW CP-DP synchronization. The OFW synchronization message format is, as shown in Fig. 3.1 (c). *indices* stores the indices of slots to mark as empty (last bit set to 0). *xor* stores a value to apply XOR with the influenced connected component in the bipartite graph, and *cc* stores the indices of the slots in the influenced connected component, i.e., a tree in G. OFW DP performs key additions and value modifications via traversing the tree to change the values in the arrays A and B trivially.

Implementation optimizations. Further optimizations are made onto the existing version of Othello hashing [105]. 1) A valid bipartite graph in an Othello is loop-free. During a value update or a key addition, we re-assign values to half of instead of all the slots in the connected component in the bipartite graph. 2) During a value update, edges in G are critical information because we have to know the connected component in order to change values by traversing. In the previous implementation, the connection information is stored by maintaining one linked list of **edges** at each node, and a hash function is invoked to find out every neighbor. We assign one linked list of **the connected edge-neighbor pairs** to each node, reducing one hash function call while keeping one memory load at each traverse step. 3) To ensure G is loop-free after a key joining, a loop detection can be performed in two possible ways: a) maintain a disjoint set during the additions and deletions and check the two slots $A[h_1(k)]$ and $B[h_2(k)]$ are in different sets; b) traverse the connected component starting from $A[h_1(k)]$ and see if $B[h_2(k)]$ is encountered. Although both ways cost $O(1)$ time, we choose approach a) because a) only involves 2 memory loads and 1 comparison, while b) requires at least 2 memory loads and more calculations. The previous implementation uses b).

3.5 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 3.1.

3.5.1 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 be the memory footprint of the most 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 $m = n_h n_k / \ln 2$ for n_h hash functions [28].

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	$e_l = \frac{1}{r_l}$
n_h	number of hash functions for a Bloom filter

Table 3.1: Notations

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. We show in Appendix A.1.3 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: $M^o_f = M^o = 2.33 n_k \cdot (l_p + l_f)$, where the coefficient 2.33 is derived in [105].

3.5.2 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. Memory accesses and hash function invocations are highly related because the main purpose of hash functions is to direct the memory access location. However, there are some differences between the two numbers: 1) other than memory loads for slots, the key itself should be loaded from the memory, which is not ignorable for many cases; 2) CFW and OFW need to hash the key for one additional time to get the fingerprint of the key; 3) the memory read for one bucket in a Cuckoo Filtable or one slot in a Cuckoo

Filtable or an Othello is not always achieved in one memory load operation because the target memory area may be too long to fit into a cache line, or it may exist in two consecutive cache lines; 4) for Level 2 of the CFW, the lengths of the keys are not constant especially for L7 so that longer keys should be put into the dynamic memory, *e.g.*, slabs and cannot be directly embedded into the slots. 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 are skipped due to space limit and are available on [7] and Appendix A.1.5

BFW.

$$\begin{aligned} E(C_h^b) &= \sum_{i=1}^{d-1} \frac{1}{d}\left((i-1)C_t + n_h\right) = \frac{d-1}{2}C_t + n_h \\ E(C_{h,e}^b) &= \left(\sum_{i=1}^{d-1} \left((1-p)^{n_h}\right)^{i-1} \cdot \left(1-(1-p)^{n_h}\right)(i \cdot C_t)\right) \\ &\quad + \left((1-p)^{n_h}\right)^d (d \cdot C_t) \end{aligned} \tag{3.1}$$

$$\begin{aligned} E(C_m^b) &= E(\lceil l_k/l_c \rceil + C_h^b) = \lceil l_k/l_c \rceil + E(C_h^b) \\ E(C_{m,e}^b) &= E(\lceil l_k/l_c \rceil + C_{h,e}^b) = \lceil l_k/l_c \rceil + E(C_{h,e}^b) \end{aligned} \tag{3.2}$$

CFW. (Assuming the key locations are uniformly random)

$$\begin{aligned} E(C_h^c) &= 2 + \frac{n_b - 1}{2} + \left(1 - E_\eta(l_f)\right)(1 + n_b) \\ E(C_{h,e}^c) &= 1 + n_b + n_b = 1 + 2n_b \end{aligned} \tag{3.3}$$

$$\begin{aligned} E(C_m^c) &= \lceil l_k/l_c \rceil + \sum_{i=1}^{n_b \cdot n_s} \frac{\lfloor i/n_s \rfloor \cdot E(C_b) + E(C_{m,i})}{n_b \cdot n_s} \\ E(C_{m,e}^c) &= \lceil l_k/l_c \rceil + n_b \cdot E(C_b) \end{aligned} \tag{3.4}$$

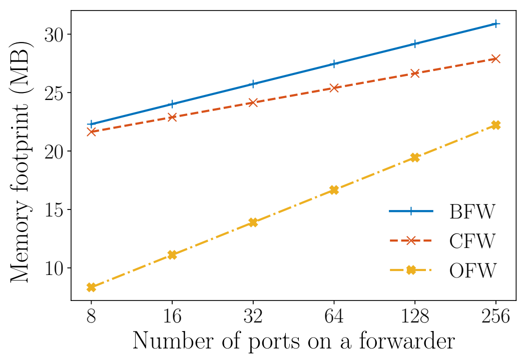


Figure 3.6: Memory with gateways

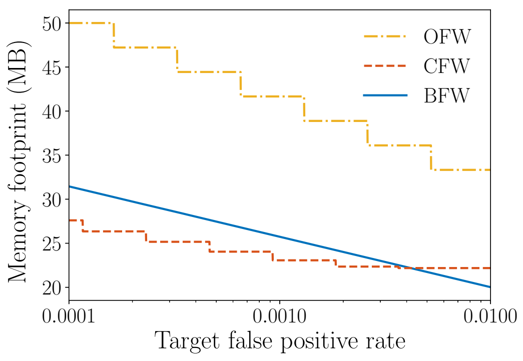


Figure 3.7: Memory w/o gateways

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:

$$\begin{aligned} C_h^o &= 3 \\ C_{h,e}^o &= \epsilon_a + 2 \cdot (1-\epsilon_a)\epsilon_b + 3 \cdot (1-\epsilon_a)(1-\epsilon_b) \end{aligned} \tag{3.5}$$

$$\begin{aligned} E(C_m^o) &= \lceil l_k/l_c \rceil + 2 \cdot \left(1 + \frac{l_f + l_p - l_g}{l_c}\right) \\ E(C_{m,e}^o) &= \lceil l_k/l_c \rceil + ((1-\epsilon_a)\epsilon_b + 2 \cdot (1-\epsilon_a)(1-\epsilon_b)) \\ &\quad \cdot \left(1 + \frac{l_f + l_p - l_g}{l_c}\right) \end{aligned} \tag{3.6}$$

3.5.3 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 to end 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, and the detailed derivation can be found in Appendix A.1.6.

BFW.

$$\begin{aligned} FP^b &= 1-(1-(1-p)^{n_h})^d \\ E(CR^b) &= \sum_{i=1}^{d}\left(1-\frac{1}{2^{n_h}}\right)^{i-1}\cdot\frac{1}{d\cdot 2^{n_h}} \end{aligned} \tag{3.7}$$

CFW.

$$\begin{aligned} CR^c &= 0 \\ FP^c &= 1-(1-\frac{1}{2^{l_f}})^{r_l E_\eta(l_f)\cdot n_s n_b} \end{aligned} \tag{3.8}$$

OFW.

$$\begin{aligned} CR^o &= 0 \\ FP^o &= \frac{1}{2^{l_f-1}}\cdot(1-\epsilon_a)\cdot(1-\epsilon_b) \end{aligned} \tag{3.9}$$

3.5.4 Numerical results and discussions

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

DP memory footprint. We consider the DP memory in two situations: with and without gateways. As described in § 3.3, 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

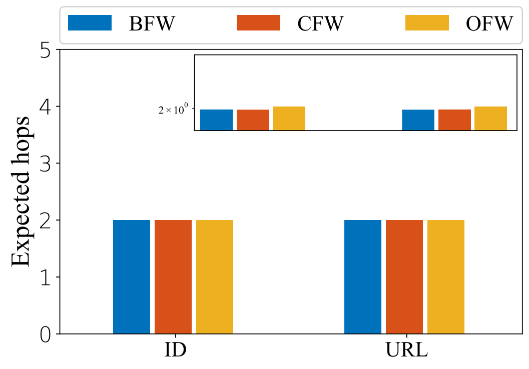


Figure 3.8: Expected forwarding hops for invalid requests

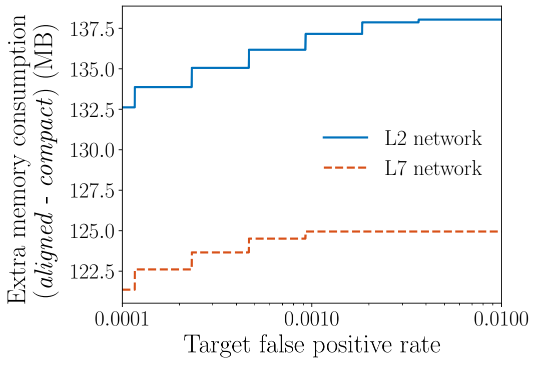


Figure 3.9: Extra memory consumption of CFW when buckets are aligned to cache lines

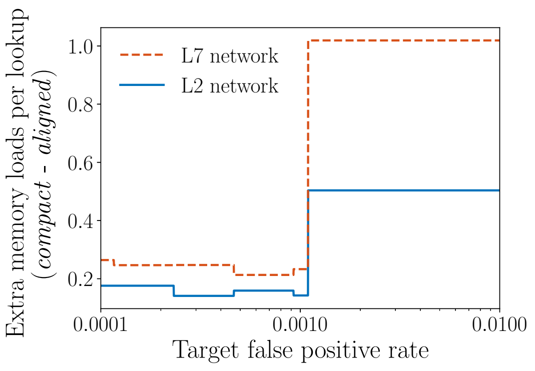


Figure 3.10: Extra memory loads per lookup of CFW when buckets are not aligned to cache lines

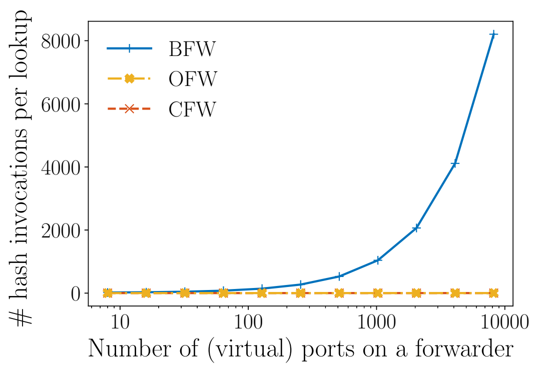


Figure 3.11: Hash function invocation time explodes with the number of ports on forwarders

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 that 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. 3.6 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, as detailed in Appendix A.1.7. The numerical results are shown in Fig. 3.7.

Comparing the results in Figures 3.6 and 3.7, it is clear that when gateways exist, OFW costs 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.

False positives in L7. In the L7 overlay network model, a client request is received by an overlay node such as a CDN node, and then the node checks the local resource pool. If the request is met, it replies with the result; otherwise, the request is forwarded to another edge node according to the FIB. Invalid requests should be dropped here because of FIB miss. Due to false positives, some requests may be wrongly forwarded to other nodes. We calculate the expected forwarding hops for valid requests of the three forwarders in the L7 network as a direct effect of false positives. We set l_f to 8 for OFW, and adjust l_f for CFW and m for BFW to let them have the same memory footprint with OFW. The result is shown in Fig. 3.8. We find that although OFW has more false positives with the same memory footprint, the impact on L7 routing path is minimal.

Cache line alignment. The above results of CFW are based on the *compact* setting, which means buckets are not aligned to cache lines, as opposed to the *aligned* setting. We compare the memory footprints as well as the expected number of memory loads between *compact* and *aligned* settings in simulated L2 and L7 networks ($l_p = 5$, $l_k = 128$ to simulate the L2 networks, and $l_p = 14$, $l_k = 360$ for the L7 networks), as shown in Figures 3.9 and 3.10. From the figures, we conclude that there is little value to align the buckets to cache lines because the latency gain is too small compared to >2x memory cost.

Hash function invocation in L2 and L7 networks. The number of hash function invocation C_h is a simple indicator for the lookup performance because hashing is more time consuming than other basic operations, and a hash value usually directs a memory load destination. The expected value and upper bound of C_h for both CFW and OFW are rather low because of their designs. The C_h for OFW is at most 3 for an alien key and is exactly 3 for a valid key. In CFW, the upper bound is 9 for both a valid key and an alien key. But C_h can be very high for BFW and is linear to d. Because on the one hand, d may be high, an alien key should be tested against all d Bloom filters to be considered as alien, and a valid key is expected to pass the correct Bloom filter after $\frac{d}{2}$ Bloom filters. And on the other hand, n_h may be large to meet a strict false positive target, especially when the d is large and the false positive rate for a single Bloom filter must be very low to effectively filter out an alien key from d Bloom filters at the FIB. The most important issue of BFW is the scalability with the number of ports. We set $n_k = 10M$, $l_k = 360$, $l_f = 8$ for both OFW and CFW, and change l_p. We clearly show the exponential explosion of C_h^b while comparing with CFW and OFW in Fig. 3.11. So it is clear that BFW is not suitable for a network where the number of neighbors may be large for a node, e.g., an L7 network.

3.6 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 the Tensorflow repository [11], 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 [4] 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 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) [5] running in CloudLab [1]. DPDK is a series of libraries for fast user-space packet processing [5]. 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 E5-2660 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 [69] 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 in the L2 network and hosts in the L7 network. Node 2 uses the DPDK official packet generator Pktgen-DPDK [6] 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 link 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.

L2 Network setup. We use one ISP network topology from the Rocketfuel project [90] 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 the 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.

L7 network setup: We model the L7 network topology as a fully connected graph with 318 nodes, the same as the number of nodes in the L2 network model. The requested keys may be valid or alien, sampled from the *std::string* representation of the resource IDs (45 bytes) and resource URLs (155 bytes). To filter out most alien requests while keeping the FIB small enough, we set $l_d = 8$ for both CFW and OFW according to the analytical results in § 3.3.

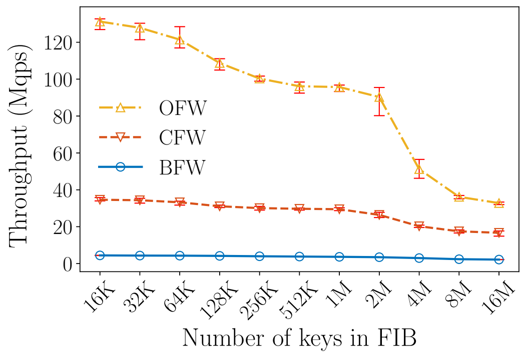


Figure 3.12: L2 DP lookup throughput for Zipfian distribution

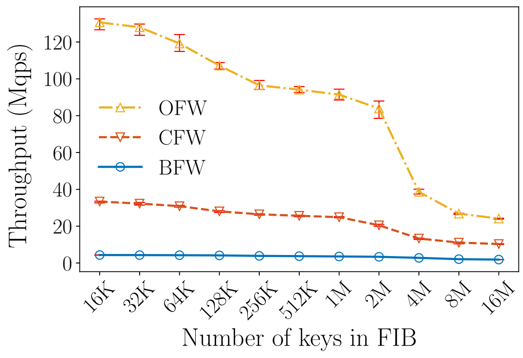


Figure 3.13: L2 DP lookup throughput for uniform distribution

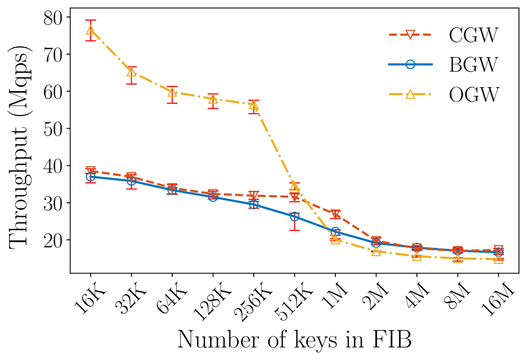


Figure 3.14: L2 gateway throughput for Zipfian distribution

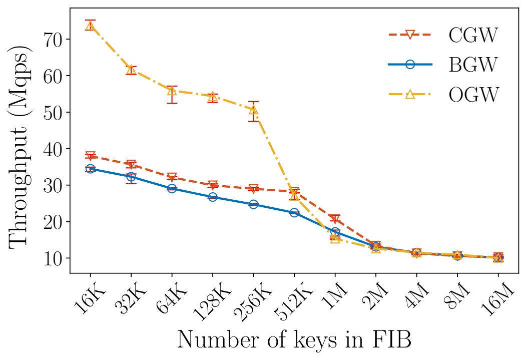


Figure 3.15: L2 gateway throughput for uniform distribution

3.7 Evaluation

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

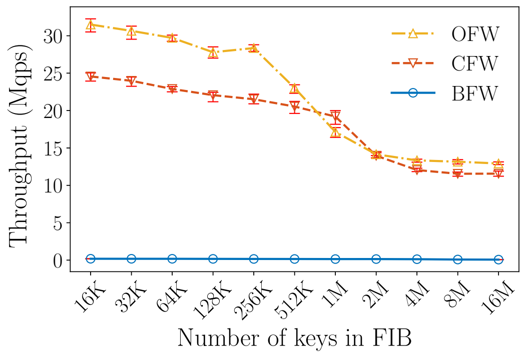


Figure 3.16: L7 DP lookup throughput for Zipfian distribution

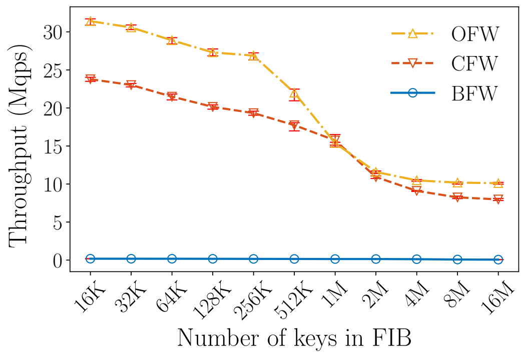


Figure 3.17: L7 DP lookup throughput for uniform distribution

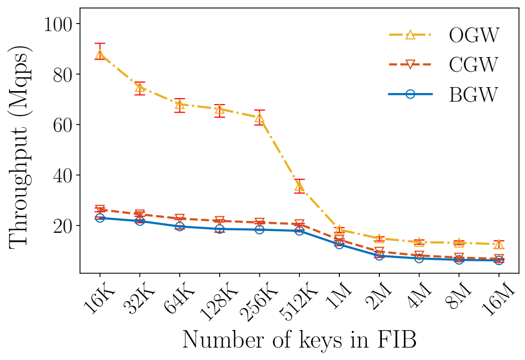


Figure 3.18: L2 gateway throughput for invalid addresses

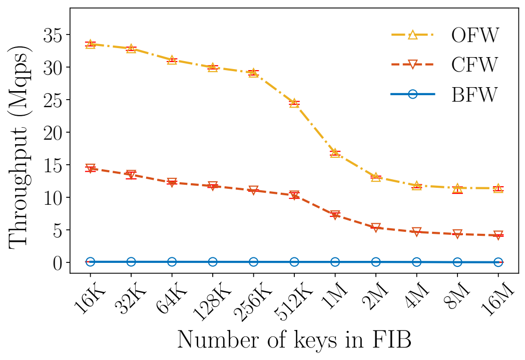


Figure 3.19: L7 DP lookup throughput for invalid addresses

3.7.1 Comparison methodology

We identify the most common situations in typical networks to abstract important parameters from these situations for comparison. We evaluate the data plane

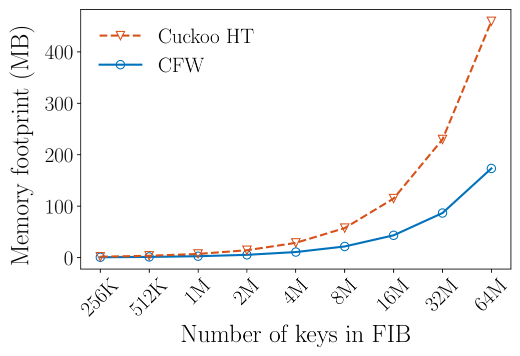


Figure 3.20: Memory: CFW vs. Cuckoo hashing

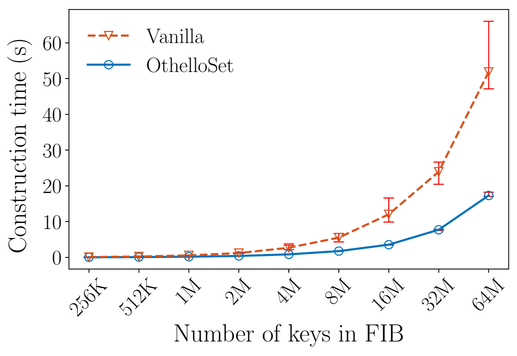


Figure 3.21: DP construction: OthelloSet vs. Concise

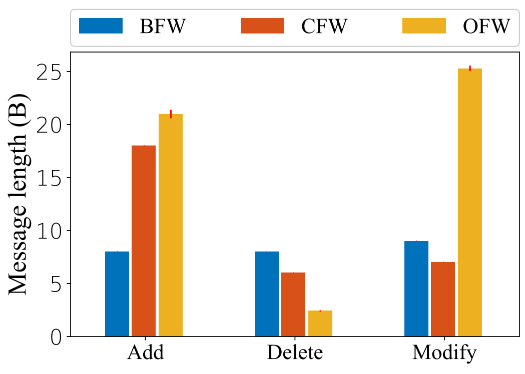


Figure 3.22: Length of incremental update messages for L2

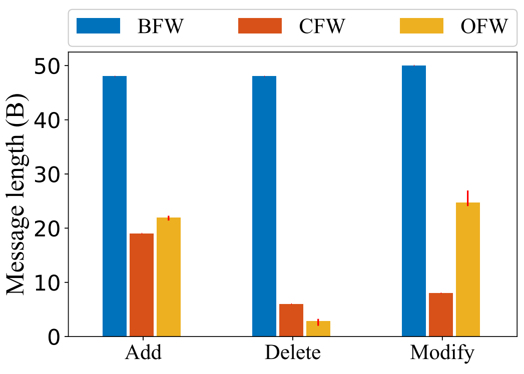


Figure 3.23: Length of incremental update messages for L7

forwarding throughput, control plane construction time, data plane construction time, and data plane incremental update throughput for all three forwarders.

The distributions of lookup requests are simulated in two types: uniform distribution and Zipfian distribution. To understand the performance variations, each data

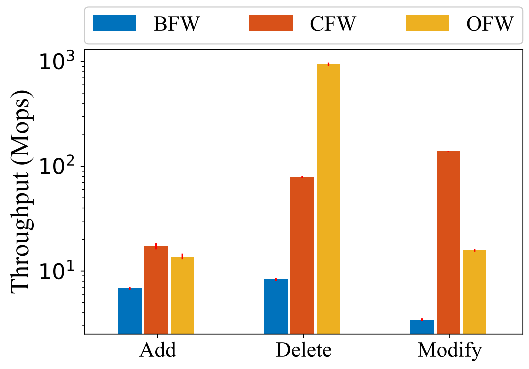


Figure 3.24: Throughput (speed) of incremental updates for L2

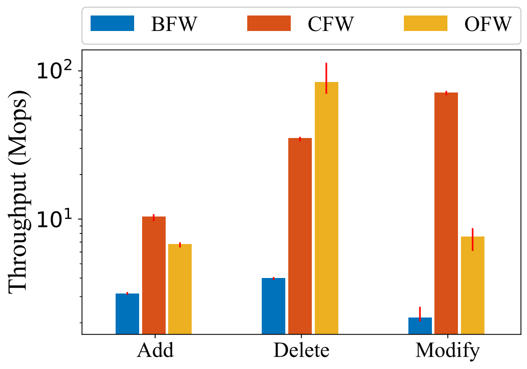


Figure 3.25: Throughput (speed) of incremental updates for L7

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 evaluate 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) Throughput of valid keys and invalid keys in L2 and L7 networks; 2) Control plane to data plane synchronization latency; 3) Control plane construction time.

3.7.2 Algorithm evaluation

Compare to prior methods. We have conducted experiments of the studied methods with prior solutions: CFW vs. CuckooSwitch [111]; OFW vs. Concise [105]. **All**

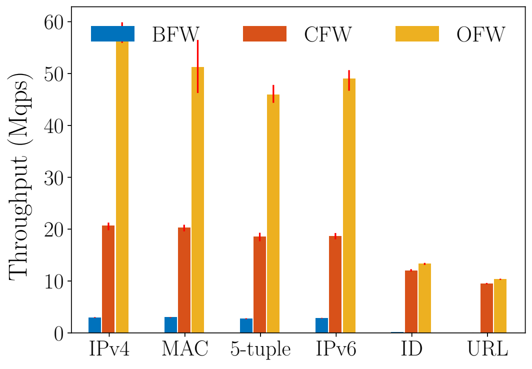


Figure 3.26: Lookup throughput for different key types

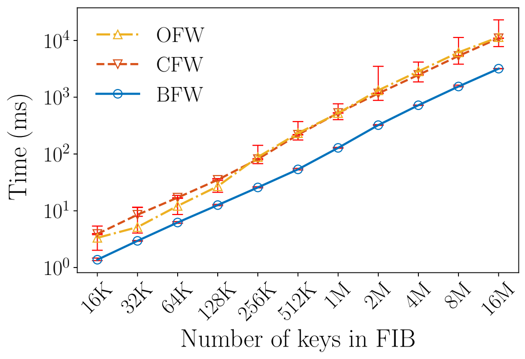


Figure 3.27: L2 network CP construction time

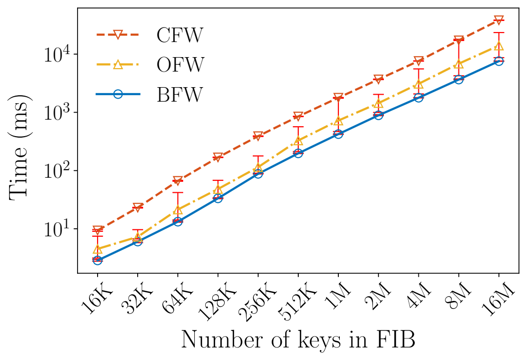


Figure 3.28: L7 network CP construction time

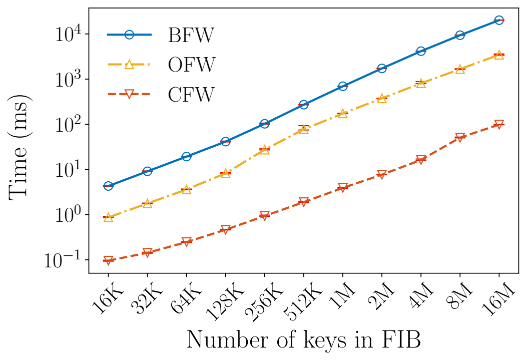


Figure 3.29: L2 network DP construction time

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. 3.20 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. 3.21, 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.

L2 throughput. We evaluate the lookup throughput of both the gateway node and core nodes in L2 networks. Figures 3.12 to 3.15 show the throughput of BFW, BFW gateway (BGW), CFW, CFW gateway (CGW), OFW, and OFW gateway (OGW) an L2 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.

L7 throughput. For L7 overlay networks, the forwarding throughput is examined with both valid addresses and alien addresses at each forwarder. Figures 3.16 and 3.17 show the throughput of BFW, CFW, and OFW in an L7 network where forwarding addresses are valid resource IDs. 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.

There is a noticeable drop in the OFW and CFW lookup throughput in the L7 network compared to those in L2. Because the hash function invocation is much slower with 10x longer input and the memory footprint of OFW in the L7 network is around 3x larger than that in the L2 network because of two reasons: 1) OFW has to store key fingerprints to filter alien addresses; and 2) the value length is changed from 8 to 16 due to more neighbors in the overlay. The throughput of BFW is only $< 10\%$ compared to the other two because BFW needs to check the Bloom filter of every neighbor (considering false positives). As shown in Fig. 3.26, with the same number of addresses, the lookup throughput decreases with the increasing length of addresses.

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. 3.26 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 in

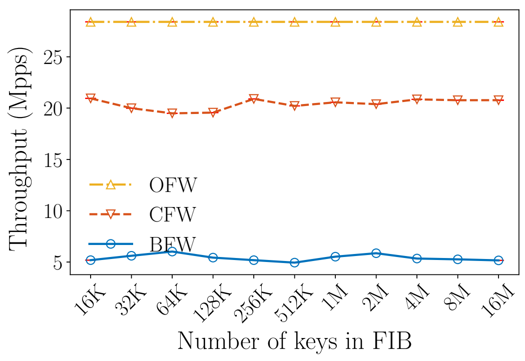


Figure 3.30: L2 DP throughput for Zipfian (single thread)

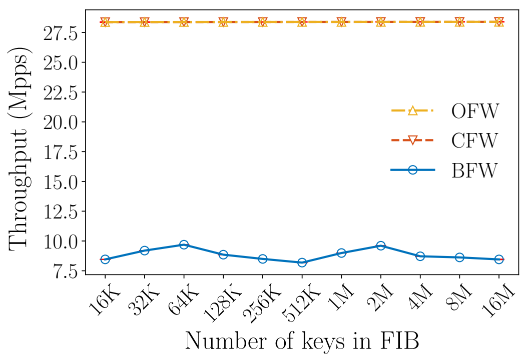


Figure 3.31: L2 DP throughput for Zipfian (two threads)

L2 networks and the alien address lookup at forwarders in L7 networks. Fig. 3.18 shows the throughput of BFW gateway (BGW), CFW gateway (CGW), and OFW gateway (OGW) where forwarding addresses are invalid MAC addresses, and Fig. 3.19 shows the throughput of BFW, CFW, and OFW in an L7 network where forwarding addresses are invalid resource IDs. We vary the total amount of addresses stored in the FIBs. All gateways show performance decreases with alien addresses because CFW performs 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. In L7 networks, CFW also exhibits a drop in throughput because the alien key lookups perform matching with 16 slots in two levels. The performance drop for OFW is caused by the memory expansion, and the decrease only happens at small FIB sizes.

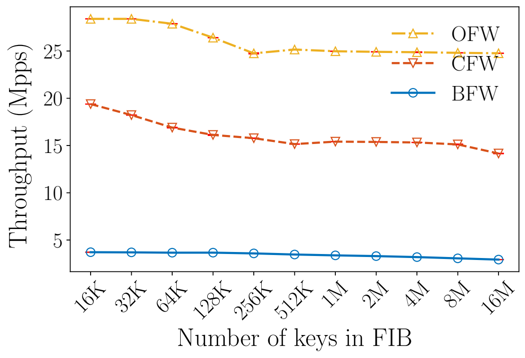


Figure 3.32: L2 DP throughput for Uniform (single thread)

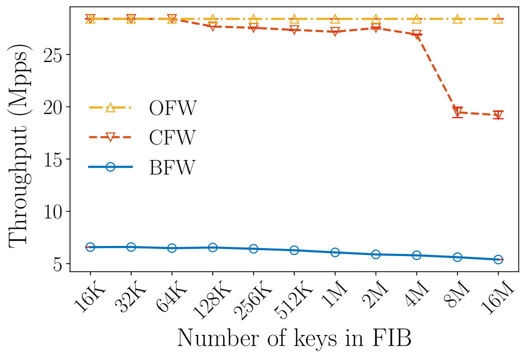


Figure 3.33: L2 DP throughput for Uniform (two threads)

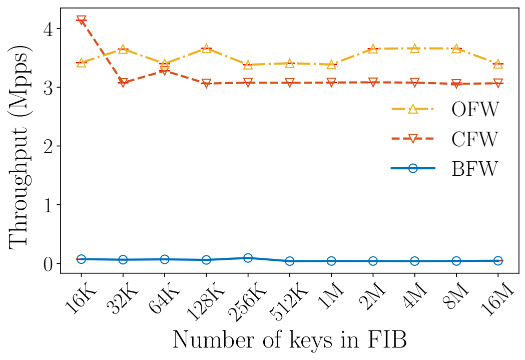


Figure 3.34: L7 DP throughput for Zipfian (single thread)

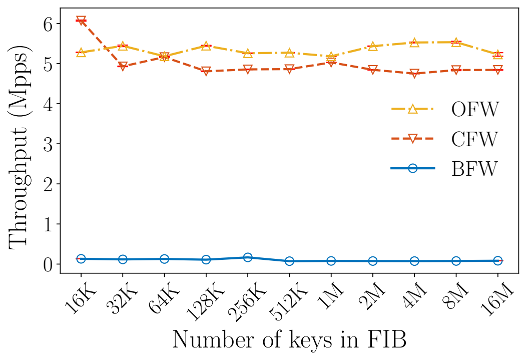


Figure 3.35: L7 DP throughput for Zipfian (two threads)

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

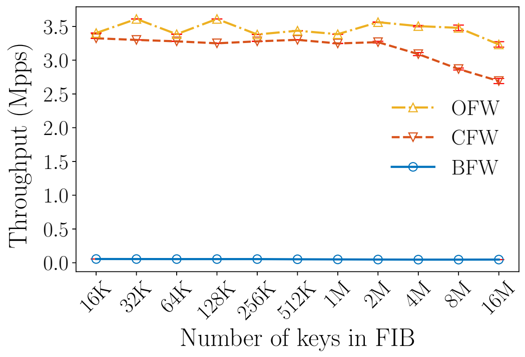


Figure 3.36: L7 DP throughput for Uniform (single thread)

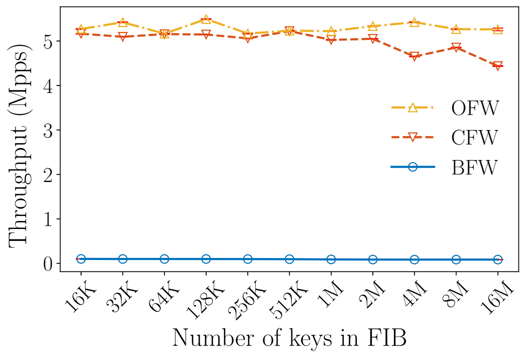


Figure 3.37: L7 DP throughput for Uniform (two threads)

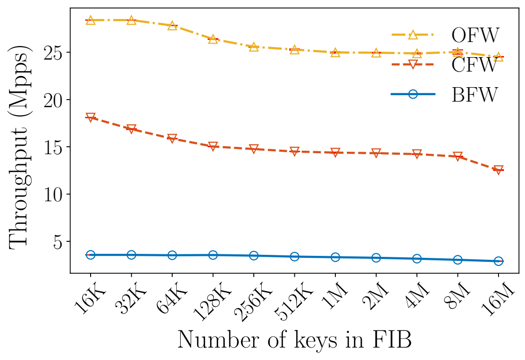


Figure 3.38: DP throughput in CloudLab (invalid MACs, 1 thread)

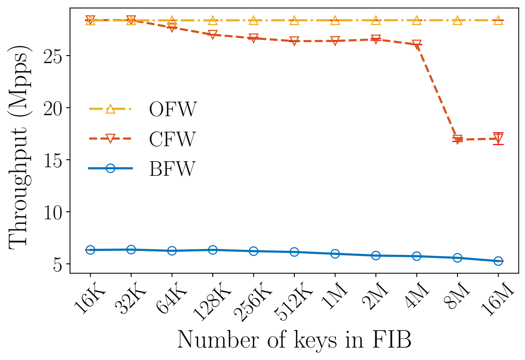


Figure 3.39: DP throughput in CloudLab (invalid MACs, 2 threads)

modeled 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

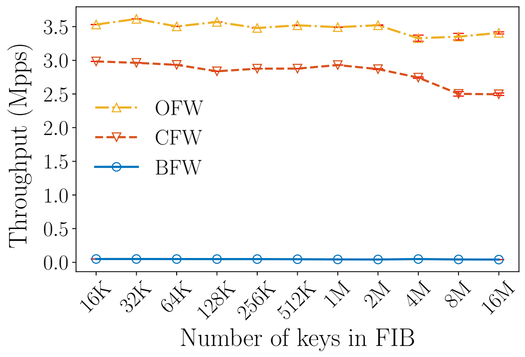


Figure 3.40: L7 DP throughput for invalid IDs (single thread)

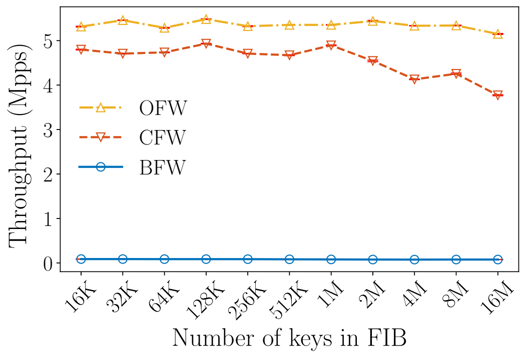


Figure 3.41: L7 DP throughput for invalid IDs (two threads)

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 for both models and use the MAC addresses for both models and use the MAC addresses as addresses in the L2 network model and IDs as addresses in the L7 network model. 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).

Figures 3.22 and 3.23 show the update message lengths of BFW, CFW, and

OFW in the L2 and L7 networks. Unlike those of BFW, the update message lengths of CFW and OFW change little in different network models because the key information is required in three types of update messages of BFW and the addition update messages of CFW only when the insertions fall into Level 2 of the FIB. 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.

Figures 3.24 and 3.25 show the update throughput of BFW, CFW, and OFW in L2 and L7 network models. 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. Figures 3.27 and 3.28 show the control plane construction time at different FIB sizes in L2 and L7 network models. The keys are MAC addresses in the L2 network model and resource IDs in the L7 network model. CFW and OFW are about 5x slower than BFW in CP construction. That is because Cuckoo Filtable is faster to construct than Othello, and the two-level design degrades 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. 3.29 shows the construction time from the CP to a single DP at different FIB sizes in the L2 network model. 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.

3.7.3 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 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.

L2 throughput. Figures 3.30 to 3.33 show the throughput of BFW, CFW, and OFW in an L2 network where the forwarding keys are valid MAC addresses. We vary the total amount of addresses stored in the FIB and observe the throughput. The addresses are sampled from both Zipfian and uniform distributions. The forwarders have lower throughput under uniform key distribution because the memory access pattern exhibits a lower locality. OFW performs the best among the three on both single thread and two threads. While the 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.

L7 throughput. Figures 3.34 to 3.37 show the throughput of BFW, CFW, and OFW in an L2 network where forwarding addresses are valid resource URLs; and Figures 3.40 and 3.41 show the throughput when addresses are all invalid. We change the total amount of addresses stored in the FIB and observe the throughput in terms of million queries per second (Mqps). Valid addresses are sampled from both Zipfian and uniform distributions. While uniform key distribution is worse than Zipfian distribution for BFW, CFW, and OFW, CFW performs the worst when addresses are all invalid because CFW is forced to try all possible slots to conclude the absence for each alien address. OFW shows the highest throughput for most experiments mainly because of its quick lookup and its independence from address validness. Other interesting observations are: 1) All three forwarders failed to reach the forwarding capacity because for each packet, they need to calculate the digest of its destination URL, which is much longer than a MAC address; 2) L7 lookup is bounded by the bus bandwidth between CPU and memory. The reason is listed as following: 1) L7 routing capacity is much larger than what the three forwarders achieve, which means the PCIe bandwidth is not the bottleneck; 2) The throughput does not grow when we add more threads for all three forwarders, which means computation is not the bottleneck.

3.7.4 Summary of comparison.

Throughput. OFW and OGW exhibit >2 times lookup throughput compared to CFW and CGW in L2 and for alien keys in L7. 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 3.5.4) 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.

Construction time. CFW and OFW are about 5x slower than BFW in CP construction but still takes < 1sec for $n = 1$M. However, the key issue is that the CP of CFW and OFW can be reused for DPs on different nodes, providing high scalability. For BFW in L2, every node needs a completely different construction process.

Performance in real networks. OFW provides higher throughput than CFW and BFW in real packet forwarding. Compared to the lookup for valid addresses, there is less throughput drop in OFW than the other two for invalid addresses. The FIB using OFW can reach full bandwidth using a single thread. As L7 forwarding involves much longer IDs than L2 forwarding, all methods cannot reach the forwarding capacity due to memory bandwidth between CPU and memory.

3.8 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 the 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 proposal, *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 § 3.4. 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) [70] 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.

Chapter 4

Ludo hashing: Compact, Fast, and Dynamic Key-value Lookups for Practical Network Systems

4.1 Overview

Key-value lookup engines running in fast memory are crucial components of many networked and distributed systems such as packet forwarding, virtual network functions, content distribution networks, distributed storage, and cloud/edge computing. These lookup engines must be memory-efficient because fast memory is small and expensive. This work presents a new key-value lookup design, called Ludo hashing, which costs the least space ($3.76 + 1.05l$ bits per key-value item for l-bit values) among known compact lookup solutions, including the recently proposed partial-key Cuckoo and Bloomier perfect hashing. In addition to its space efficiency, Ludo hashing works well with most practical systems by supporting fast lookup, fast updates, and concurrent writing/reading. We implement Ludo hashing and evaluate it with both micro-benchmark and two network systems deployed in CloudLab. The results show that in practice, Ludo hashing saves 40% to 80%+ memory cost compared to existing dynamic solutions. It costs only a few GB of memory for 1 billion key-value items and achieves high lookup throughput: over 65 million queries per second on a single node with multiple threads.

4.2 Related Work

In-memory key-value lookup engines with small memory footprint support vital functions of many networked and distributed systems, including network forwarding [104, 111, 107, 110], distributed storage [76, 97], cloud load balancers [67], and content distributions [43, 65]. Space efficiency is the most significant requirement of these applications because they are all running in fast and small memory, such as cache, DRAM, or ASICs, in order to serve frequent lookups.

Hash Tables are conventional tools for in-memory key-value lookups. Most existing hash table implementations require storing the complete keys. In particular, Cuckoo Hashing [74] could achieve $O(1)$ lookup time in the worst case and amortized $O(1)$ update time. The construction and lookup of the (2,4)-Cuckoo hashing [74] is introduced in § 2. Many recent system designs choose the (2,4)-Cuckoo to achieve high memory utilization and fast lookups, such as the memory cache system MemC3 [40], the software switch CuckooSwitch [111], the LTE FIB ScaleBricks [110], and the cloud load balancer Silkroad [67]. The *amortized* insertion time of (2,4)-Cuckoo is proved to be constant [68, 98] and empirically shown [74, 40]. The insertions are proved to be successful asymptotically almost surely (a.a.s.) for load factor $< 98.03\%$ and $n \to \infty$ [31, 45].

Partial key Cuckoo hashing (PK Cuckoo) costs less space by storing the key digests instead of full keys. A basic version of PK Cuckoo is proposed in [41], and a more compact version, Vacuum filter, is proposed in [95]. SILT [64], an index for flash storage, proposes to use 15-bit key digests instead of the full keys. Using key digests is not a trivial solution. Short key digests incur hash collisions and false mappings, and a nontrivial two-level design is proposed in [87] to address the collisions.

EMOMA [79] is a lookup data structure with a full version of (2,4)-Cuckoo holding the key-value mappings. A counting block bloom filter (CBBF) is placed in the cache to maintain the bucket choice of each key, such that each lookup costs exactly one off-chip memory load. There are three major differences between Ludo and EMOMA: 1) Ludo aims to reduce the memory cost while EMOMA requires significantly more memory cost – even higher than a full (2,4)-Cuckoo. The key reason is that Ludo resolves collisions within a bucket via a very short seed instead of storing full keys in EMOMA. 2) EMOMA optimizes the lookup throughput while Ludo does not. 3) Ludo records the bucket choice of all keys without any error, while EMOMA uses a CBBF, which exhibits false positives and counter overflows. 4) On a single insertion, keys in EMOMA may be inserted into and deleted from the CBBF multiple times, which hurt the update speed.

Bloomier filters [35, 34] are instances of minimal perfect hashing (MWHC) [26, 66, 35, 34, 107], originally proposed for static lookup tables. Othello Hashing is a data structure and a series of algorithms based on Bloomier filters designed for dynamic forwarding information bases [107]. Othello hashing includes both the lookup structure running in fast memory such as switch ASICs and a maintenance structure running in resource-rich platforms such as servers. The construction and lookup of Othello hashing is introduced in § 2. Coloring Embedder [101] is a recent work with a similar design to Bloomier. Its space cost is also close to that of the Bloomier and Othello.

SetSep [42, 110] is a lookup table that uses brute force to resolve collisions. Suppose the key set has cardinality n, and all values are of the same length l. During SetSep construction, a global hash function distributes the keys across $\lceil n/4 \rceil$ buckets, each of which contains 4 keys *on expectation*, with high variations. 256 consecutive buckets form a block, and blocks are built independently. To build a block, a greedy algorithm is used to map its buckets to 64 groups, each holding 16 keys *on expectation*. For the i-th value bit in each group, a 16-bit array m and an 8-bit hash seed s are found by brute-force, such that for every key value pair (k, v) in the group, $m[h_s(k)] = v_i$, where $v_0, v_1, \cdots, v_{l-1}$ are bits of v. All key-value items of the failed groups are put into a small plain hash table. Ludo hashing provides two major advantages over SetSep. First, Ludo hashing can be updated in $O(1)$ complexity, while a single insertion into SetSep may cause reconstructions of the involved group, block, or even the whole data structure. The main challenge of its updates is that SetSep has no theoretical or empirical bound on the average number of group/block/global level reconstructions per update. Experimental results show that SetSep takes 10x construction time and >1000x update time compared to Ludo. Second, Ludo hashing has the smallest space cost among the aforementioned algorithms when the value length is > 7, which is the case for most applications.

4.3 Problem Definition and Models

We formally define the problem in this work. We are given a set of key-value items S and $|S| = n$. Each item in S is a tuple $\langle k_i, v_i \rangle$ of key k_i and value v_i. Every key is unique in S. All values have the same size (i.e., number of binary digits), denoted by l. The goal of this work is to find a key-value lookup engine that provides the following functions with minimized time and space costs.

1. The lookup function `query`(k) returns the corresponding value v for the query key k, where $\langle k, v \rangle \in S$.

2. The construction function `construct`(S) constructs a table for the set S.

3. The insertion function `insert`(k, v) inserts the item $\langle k, v \rangle$ to the current table.

4. The deletion function `delete`(k) deletes the item with key k from the current table.

5. The value change function `remap`(k, v') changes the value of the item with key k to v', in the current table.

System model. The proposed Ludo hashing includes the *lookup structure* and *maintenance structure.*

- The lookup structure in fast memory focuses on the lookup function. Its space cost and lookup time are minimized.
- The maintenance structure maintains the full key-value state and performs construction and update functions. It can run in a different thread or even on a different machine from where the lookup structure runs.
- Necessary update information will be constructed by the maintenance structure and sent to the lookup structure. The time complexity of each update is an important metric.

For space-efficient lookup engines that do not store full keys, a separate maintenance structure is *necessary* to support updates. Otherwise, update correctness cannot be guaranteed. In practice, the lookup structure is hosted in fast and small memory, while the maintenance structure can be hosted in slower but larger memory. This model has been extensively used in system designs [64, 60, 43, 65, 107, 108, 67, 110, 42].

4.4 Design of Ludo hashing

4.4.1 Challenges and the main idea

A typical MPHF consists of two-level hashing [27]. The first level hashing $g : U \to [0, r-1]$ divides the entire set K of n keys randomly into r buckets. The numbers of keys in all buckets vary significantly, and the maximum number of keys in a bucket is much bigger than n/m based on the 'balls into bins' results [82]. The buckets are sorted in descending order of their size. In this order, the second level finds a hash function $f_i : U \to [0, m-1]$ for each bucket B_i such that the hash result of every key in B_i does not collide with any other key in all previous buckets. Let $\epsilon = m/n - 1$ and $\lambda = n/r$, the time complexity of the above construction is $O(n(2^\lambda + (1/\epsilon)^\lambda))$ [27]. In most cases, an insertion will cause the reconstructions of $O(r)$ second level hashes f_i.

Our main contribution is to allow each update to finish in $O(1)$ time by a novel utilization of (2,4)-Cuckoo and Othello, which has not been discovered before. Ludo first uses (2,4)-Cuckoo and Othello together to build a function F that divides the keys into r buckets, each of which has up to 4 keys, and then find a seed to resolve the collisions among each bucket. This design provides two unique benefits: 1) each insertion only affects $O(1)$ buckets (proved by [68, 98] and empirically < 6 among all our experiments), while in other MPHFs, this number is unbounded; 2) within each bucket, Ludo only needs to find a hash that maps four keys to $[0, 3]$ without collision, which is significantly easier than other MPHFs that need the results to be collision-free across all buckets.

Step 1: Uniform-sized grouping. By observing the (2,4)-Cuckoo Hash Table shown in Fig. 2.2, we find that it includes a number of buckets, each containing up to 4 keys. This organization is close to our requirement of uniform-sized grouping. However, each key could be placed to any of its two alternate buckets based on the insertion process of (2,4)-Cuckoo. If we use a simple hash function to map keys to buckets, then the sizes of buckets suffer from a high variation. Resolving collisions of different numbers of keys will cause a significant waste of space, as shown in § 4.4.6. Hence, our idea is to combine a Bloomier filter [33, 107] and the bucket information of keys in the (2,4)-Cuckoo as the uniform-sized grouping function F. In a (2,4)-Cuckoo, each key k can only stay in one of the two alternate buckets, indexed $h_0(k)$ and $h_1(k)$. Given an already constructed (2,4)-Cuckoo, we only need a Bloomier filter to maintain only 1 bit of information per key: whether the key stays in the bucket $h_0(k)$ or $h_1(k)$. Recall that Othello Hashing is a dynamic extension to the original Bloomier filter, and Othello supports key-value lookups *with 100% correctness* using 2.33 bits per key for 1-bit values [107]. Hence, we need 2.33 bits per key to locate each key to the bucket holding it in a constructed (2,4)-Cuckoo.

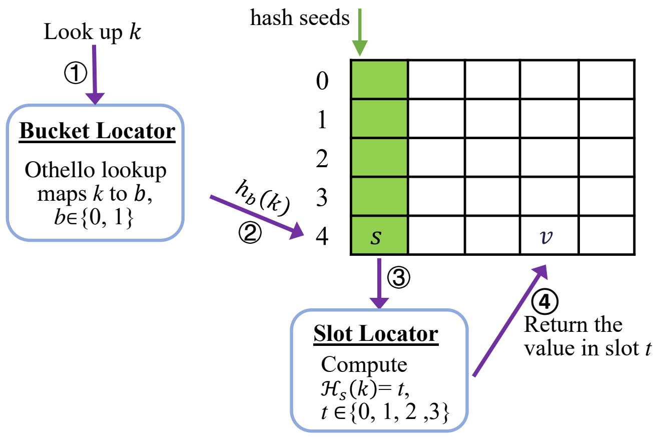
Step 2: Collision resolution. Given a bucket B of four keys, we want to find a function F'_B that maps the four keys to four different slots without collision. In this way, we can match all keys to their corresponding values without storing keys. Note that we may sample sufficiently many independent random hash functions from a universal hash function family $\mathcal{H}$. For example, Google's Farm Hash [2] accepts a 'seed' as input, and different seeds will result in independent hash functions. The probability that a randomly seeded hash function maps 4 keys to 4 slots without collision is $4!/4^4 = 3/32$. Therefore, by trying different hash functions with *brute force*, we can find a hash function that maps the 4 keys without collision in a limited number of attempts. Once a function is found for a bucket, the seed value is stored along with the bucket. In our implementation, the seed costs 5 bits, i.e., 1.25 bits per key — a significant space saving comparing with storing the keys.

4.4.2 System overview

The complete Ludo hashing includes two components: the Ludo lookup structure and Ludo maintenance structure. The Ludo lookup structure, considered as the data plane, runs in fast memory and supports lookup queries. The Ludo maintenance structure, considered the control plane, can run in slower memory, possibly on a separate machine. The lookup structure receives update information from the maintenance structure and updates accordingly.

Ludo lookup structure. As shown in Fig. 4.1, a Ludo lookup structure is a tuple $\langle O, B, h_0, h_1, \mathcal{H}\rangle$ where B is an array of buckets, each bucket $B[i]$ includes a hash seed s and 4 slots storing up to 4 values; h_0 and h_1 are two uniform hash functions; O is an Othello lookup structure that returns 1-bit value to indicate whether a key k is mapped to bucket $h_0(k)$ or $h_1(k)$; and $\mathcal{H}$ is a universal hash function family. The query of a key k will output the value v_k. Ludo lookup structure will query two locators in turn: the **bucket locator** to indicate the bucket that stores the value, and the **slot locator** to determine the slot that stores the value. The bucket locator will lookup k in Othello and get a result $b \in \{0, 1\}$. Then v is in bucket $h_b(k)$. The slot locator computes $t = \mathcal{H}_s(k)$ where s is the seed stored in this bucket and $t \in \{0, 1, 2, 3\}$. Finally, the value in slot t of bucket $h_b(k)$ is returned as v_k.

Ludo maintenance structure. As shown in Fig. 4.2, a Ludo maintenance structure is composed of two main parts: 1) a complete (2,4)-Cuckoo holding all in-



Look up k
①
Bucket Locator
Othello lookup maps k to b, $b \in \{0, 1\}$
$h_b(k)$
②
hash seeds
0
1
2
3
4
s
v
③
Slot Locator
Compute $\mathcal{H}_s(k) = t$, $t \in \{0, 1, 2, 3\}$
④
Return the value in slot t

Figure 4.1: Lookup workflow of Ludo lookup structure

serted key-value items, and each bucket stores a seed for the slot locator; 2) an Othello maintenance structure that stores whether each key is in bucket $h_0(k)$ or $h_1(k)$. It can produce an Othello lookup structure used in the bucket locator. The seed s is found by brute force such that $\mathcal{H}_s$ maps the keys in the bucket to different slots without collision. We name the full (2,4)-Cuckoo as the *'source Cuckoo table'* of the lookup structure. To generate the Ludo lookup structure, the maintenance program first generates an Othello lookup structure and sets it as the bucket locator. Then it builds a table where each bucket includes the seed and only the four values in the order of the $\mathcal{H}_s(k)$. The Ludo maintenance structure supports updates including item insertions, deletions, and value changes (Sec. 4.4.8) and will reflect them in the lookup structure. Multiple Ludo lookup structures can be produced from and associated with the maintenance structure to receive update messages and update locally.

We define the *load factor* of a Ludo hashing as the number of slots storing values to the number of total slots. We use load factor 95% as the target load factor of

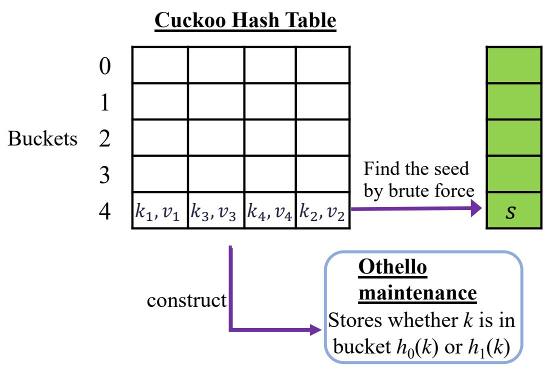


Figure 4.2: Ludo maintenance structure

Ludo. The total space cost of Ludo hashing is $3.76 + 1.05l$ for l-bit values.

4.4.3 Ludo lookup structure

We show the pseudocode of the Ludo hashing lookup algorithm in Algorithm 1. This algorithm is simple and fast. It contains two steps: querying the bucket locator and the slot locator, respectively. Each step takes $O(1)$ time.

4.4.4 The bucket locator

The bucket locator, implemented with an Othello lookup structure, maintains the bucket location of all inserted key-value items and serves in the *uniform-sized grouping* step. Given a query key k, the Ludo lookup structure locates k to a bucket by querying Othello. The return value b is 0 or 1, denoting the value of k is stored in the first alternate bucket $h_0(k)$ or the second one $h_1(k)$. The proposed bucket locator has the following properties.

1) It locates every inserted key-value item to the bucket holding it without

```
Input: The Ludo lookup structure and the key $k$
Output: The lookup result $v$ of $k$
begin
    // Step I: compute bucket location
1   $b \leftarrow$ Othello lookup result of $k$
2   $B \leftarrow h_b(k)$-th bucket of the table
    // Step II: compute slot location
3   $s \leftarrow$ seed stored in $B$
    // $\mathcal{H}_s(k) \in \{0,1,2,3\}$
4   $v \leftarrow B.slot[\mathcal{H}_s(k)]$
end
```

Algorithm 1: Ludo hashing `lookup` algorithm

error.

2) It costs amortized $O(1)$ time for dynamic updates, at high throughput in practice (over 10 million operations per second [87]). During updates, it still supports fast lookup [107].

3) The current design is a good tradeoff among solutions that are fast in lookup and updates and compact in mapping keys to $\{0,1\}$, such as SetSep [42] and Bloom filter cascades [60].

We compare their space costs in Fig. 4.3. Note that the filter cascades [60] cost different space when the distribution of keys to 0 and 1 changes. We collect the statistics of a (2,4)-Cuckoo with 100 million keys from 10 independent runs. The distribution of items stored in the bucket $(h_0(k), h_1(k))$ is $(0.7175, 0.2825)$ with the standard deviation

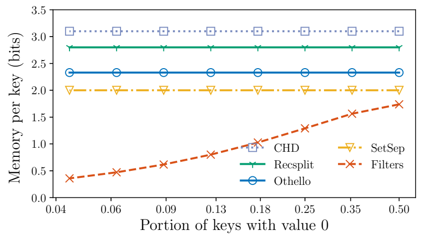


Figure 4.3: Space cost of different algorithms as the bucket locator

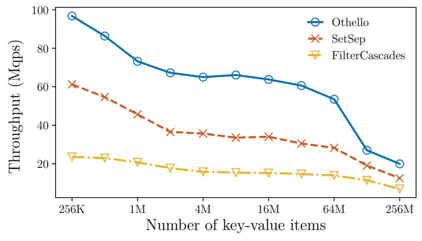


Figure 4.4: Lookup throughput of different algorithms as the bucket locator

of 0.0008. By looking at 0.28 in Fig. 4.3, SetSep and filter cascades cost less space than Othello by about 0.3 bits per key. The reason for choosing Othello is that SetSep is difficult to update, as shown in § 4.6, and filter cascades are slow in lookup because each lookup costs a higher number of memory loads on average. Fig. 4.4 shows the lookup throughput for different numbers of key-value items, where each key is a 32-bit integer, and each value is 0 or 1 at the probability 0.7175 or 0.2825, respectively. Perfect hashing algorithms like CHD [27] and RecSplit [39] are also compared here, but they are not compact enough because an additional bit array is required to store the values, which costs 1 bit per item.

4.4.5 The slot locator

After locating the bucket, Ludo hashing retrieves the bucket content that includes a seed s and 4 value slots. Ludo hashing then calculates $\mathcal{H}_s(k)$ and gets a result in range $\{0, 1, 2, 3\}$. $\mathcal{H}$ is a universal hash family, and each seed produces an independent

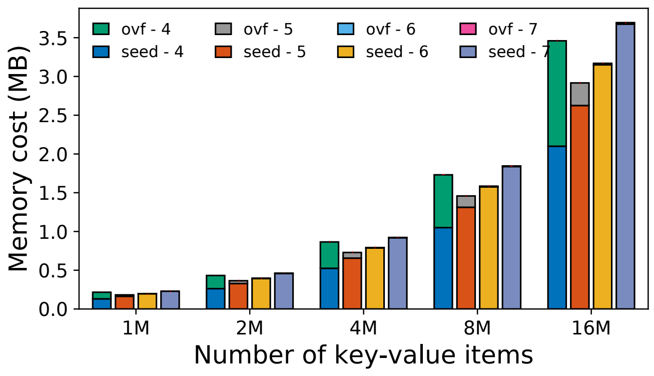


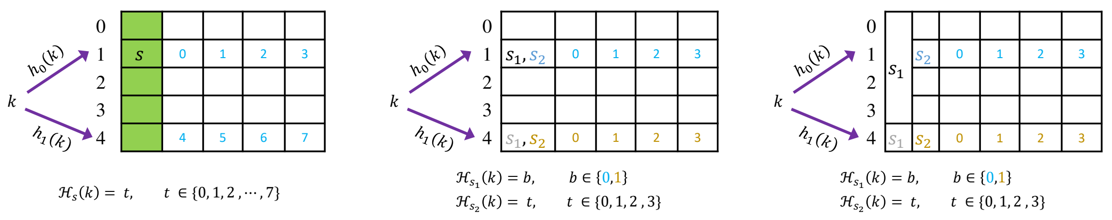
Figure 4.5: Memory cost for different seed lengths (95% load)

random hash function. Finally, it returns the value stored in the $\mathcal{H}_s(k)$-th slot.

It should be noted that the order of the values in each bucket of a Ludo lookup structure **does not necessarily follow the order in the source Cuckoo table** of the Ludo maintenance program. The order of the key-value items in a bucket of the source Cuckoo table is determined by the insertion and relocation processes. In Ludo lookup structure, however, we only need a **collision-free key-to-slot mapping**, and the order of keys makes no difference.

The brute-force seed searching starts from $s = 0$. It increases s by 1 at each time until $\mathcal{H}_s(\cdot)$ maps the 4 keys of the bucket to $\{0, 1, 2, 3\}$ without collision (called a *valid seed*). This design is much less complex than finding the seed that produces the same order of the items in the source Cuckoo table. Our experimental studies show that it saves around 4.6 bits per key and use 4.2% time.

For $O(1)$ time lookups, each bucket should have the same size. Hence, the space to store the seed in every bucket should also be the same. For e-bit seed space, if the brute-force searching cannot find a valid seed by up to value $2^e - 2$, the seed space


k
$h_0(k)$
$h_1(k)$
s
$\mathcal{H}_s(k) = t, \quad t \in \{0,1,2,\cdots,7\}$
s_1, s_2
$\mathcal{H}_{s_1}(k) = b, \quad b \in \{0,1\}$
$\mathcal{H}_{s_2}(k) = t, \quad t \in \{0,1,2,3\}$

(a) Look up a key in *Single* **(b) Look up a key in *Separate*** **(c) Look up a key in *Grouped***

Figure 4.6: Possible Ludo variants: *Single*, *Separate*, and *Grouped*

will store $2^e - 1$ (i.e., all 1 bits) to indicate that it is an *overflow seed.* Overflow seeds will be stored in a separate but much smaller table. We show the memory cost breakdown of seeds in buckets and the overflow table for different seed lengths in Fig. 4.5. Our implementation uses 5-bit seeds for minimal space cost.

The bucket and slot locators in total use 3.76 bits per key, including 2.33 bits for the bucket locator, 1.31 bits for the slot locator (assuming 95% load factor), and 0.12 bits for the overflow table. Each lookup takes 4 hash function calls and 3 memory loads — small constant time.

4.4.6 Design optimizations

The current design of Ludo lookup structure is chosen from a number of variants that achieves similar tasks, as shown in Fig. 4.6. We show the current design is more optimized than the others in the following.

Recall that each key can be mapped to two alternate buckets $h_0(k)$ and $h_1(k)$. For each bucket B, we define the '*T0 keys*' of B as the keys whose $h_0(k)$ buckets are B and the '*T1 keys*' as the keys whose $h_1(k)$ buckets are B.

Design option 1: Single locator ('*Single*'). We do not use the bucket locator. At each bucket, a hash seed is stored. For each key k, we always retrieve the seed s stored in the bucket $h_0(k)$. If the value of k is stored in the bucket $h_0(k)$, $\mathcal{H}_s(k)$ should be the correct slot position from 0 to 3. If the value is in the bucket $h_1(k)$, $\mathcal{H}_s(k)$ should be from 4 to 7, indicating one of the 4 slots in bucket $h_1(k)$. Hence, the seed s of bucket B is used for all T0 keys of B.

This method is simple to implement and requires fewer memory loads for each lookup: only one memory load with 71.75% possibility versus 3 for Ludo. However, the numbers of T0 keys of all buckets are not uniformly distributed and could possibly have high variation. In our experiments of 100 million items, some buckets may have > 20 T0 keys, and thus the brute force process could be very time-consuming and result in very long seeds. This introduces a **dilemma**: setting a short seed length leads to a large portion of seed overflow while setting a long seed length incurs big memory waste.

Design option 2: Separate seeds ('*Separate*'). This method stores two hash seeds s_1 and s_2 in each bucket: $\mathcal{H}_{s_1}(\cdot)$ computes an 1-bit value for all tier-1 keys, indicating whether the key is in bucket $h_0(k)$ or $h_1(k)$; and $\mathcal{H}_{s_2}(\cdot)$ maps all keys in this bucket to 4 slots without collision. Hence, s_1 works as the bucket locator, and s_2 works as the slot locator. Compared to the Ludo hashing design, it moves the time and space costs of Othello to the calculation of $\mathcal{H}_{s_1}(\cdot)$ and the storage of s_1. However, s_1 still needs to handle T0 keys with large variations.

Design option 3: Grouped buckets ('*Grouped*'). This method applies an additional optimization to save space for the seed s_1 in *Separate*. We combine the

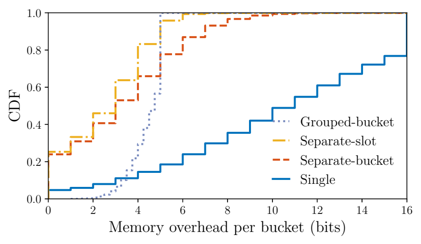


Figure 4.7: Storage overhead per bucket for different design choices (1M keys, 95% load)

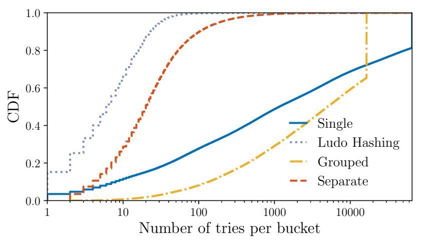


Figure 4.8: Seed attempts per bucket for different design choices (1M keys, 95% load)

space of 4 consecutive buckets as a group and use a shared space for their s_1 seeds (4 is a number chosen for good cache locality and space saving). The shared space is used to store a long seed to filter all T1 keys in all 4 buckets. This method is designed to amortize the large variation of T0 keys in every bucket.

Design comparisons. We conduct the experiments of the 4 design choices *Single*, *Separate*, *Grouped*, and Ludo hashing, and compare their results. We generate 1 million uniformly distributed 32-bit integers as keys and set the load factor of Ludo hashing to 95%. To make the evaluations finish in a reasonable time, we set an upper bound 2^{16} for the number of seed attempts per bucket. We denote the seed length for the bucket locator of *Separate* as '*Separate-bucket*', the seed length for the slot locator of *Separate* as '*Separate-slot*', and the seed length per bucket for the bucket locator of *Grouped* as '*Grouped-bucket*'. Note that the seed lengths of the slot locators of Ludo hashing and *Grouped* are both equal to *Separate-slot*.

Fig. 4.7 shows the cumulative distribution of the memory overhead (seed size) of each bucket, and Fig. 4.8 shows the number of attempts to find the right seeds for each bucket. *Single* requires much longer seed sizes and higher computation overhead than other solutions. Note that *Grouped* fails to construct more than 30% groups in 2^{16} attempts, as shown in Fig. 4.8. The sudden increase of the *Grouped* curve indicates the bound of this design. For *Separate* to work, the seed of the bucket locator requires 8 bits, allowing a small portion of overflow. This cost is thus about 2.11 bits per key, slightly less than using Othello. However, as shown in Fig. 4.8, *Separate* takes 3x time to compute the seeds compared to Ludo. Hence we believe the current design selects a good tradeoff.

Overflow seeds. As shown in Fig. 4.7, more than 98% slot locator seeds can be stored in 5 bits. Hence we set the seed length in each bucket to 5 bits. If a seed is larger than 30, it is marked overflow by storing the seed as 31. The map from the bucket index to the overflow seed is inserted into a small (2,4)-Cuckoo, called the *overflow table*, both in the maintenance structure and the lookup structure. According to the experiments in § 4.6, we show two facts: 1) We have never observed any seed that needs more than 8 bits. Hence, the value length is just 1 byte in the overflow table; 2) The overflow rate is always around 1.2% and independent from the number of items in the table. The amortized cost of overflow seeds is around 0.12 bit per key.

Insertion fallback table. Recall we set the target load factor of the source Cuckoo table to 95%, which is a load factor in our experiments that never introduce a single insertion failure in breadth-first search (BFS) within 5 steps. For the strong robustness as a system, we set aside another small hash table to store the full key-value mapping for all items failed to be inserted, although in practice we have never seen failed insertions during the experiments for load factor $< 95\%$. The fallback table is similar to the stash approach used in Cuckoo [58, 57]. We store a fallback bit along with h_a and h_b. At the beginning of each lookup, if the fallback bit is 1, it means the fallback table stores some items. Hence the fallback table is first queried, and the corresponding value is returned if there is a match. If the fallback bit is 0, the query goes through the normal lookup procedure, as shown in 4.1. In theory as long as the load factor $< 98.03\%$, the insertions are successful asymptotically almost surely (a.a.s.) assuming $n \to \infty$ [45, 31] as explained in Section 4.4 and Appendix A.2.2. This is the main reason why we never encounter a single insertion failure during our experiments. When the load factor reaches an application-dependent threshold (such as 94%) during system execution, the Ludo maintenance program will start to build a new Cuckoo table with a higher capacity, which will be used to replace the original lookup table as soon as its load factor exceeds 95%. The implementation of this fallback table can be standard hash tables such as C++ unordered_map. The rebuild happens in the maintenance server, not on the query devices.

Why (2,4)-hash table? We conclude (2,4) is the best configuration for Ludo, based on the following reasons. 1) (2,4)-Cuckoo is almost optimal in load factor (maximum load $\approx$ 98% in theory [45, 31] and $\geqslant$ 96% in practice). 2) (2,4)-Cuckoo minimize the space costs of the bucket and slot locators. Recall the bucket locator costs $2.33\lceil \log_2 d \rceil$ bits per key, where d is the number of alternative buckets. Any increment in d will cost at least 2.33 bits per key, over 60% of the current overall overhead 3.72

bits per key. Besides, 5 or more slots in one bucket contribute little to the load factor [45, 31], but the expected number of slot locator tries grows from $\sim 4^4/4! \approx 10.7$ to $\sim 5^5/5! \approx 130$ or even higher.

4.4.7 Ludo hashing construction algorithm

We design the Ludo maintenance structure to support fast construction and updates to the Ludo lookup structure. The construction takes $O(n)$ for n key-value items, and each update takes amortized $O(1)$ time.

As shown in Fig. 4.2, the Ludo maintenance structure includes 1) a (2,4)-Cuckoo, which maintains all the inserted key-value items and decides their key-to-bucket mapping; 2) a seed in each bucket to determine the slot positions of the values; 3) an Othello maintenance structure to keep track of the current Othello lookup structure. As shown in Fig. 4.9, constructing a Ludo maintenance structure and Ludo lookup structure from scratch consists of the following steps.

Step 1. We start a standard (2,4)-Cuckoo construction. All key-value items are serially inserted into the Cuckoo table, whose size is estimated by a load factor 0.95.

Step 2. For every bucket, a valid seed s is one that hashes keys to slots without collision. Numbers $0, 1, \cdots, 30$ are tested in sequence to see if any is a valid seed. If all s from 0 to 30 are invalid, the algorithm stores 31 to indicate an overflow.

Step 3. For every key, get the 1-bit bucket placement information: 0 indicates the item is stored in bucket $h_0(k)$, and 1 indicates it is stored in bucket $h_1(k)$. The algorithm then constructs the Othello maintenance structure O to track this information

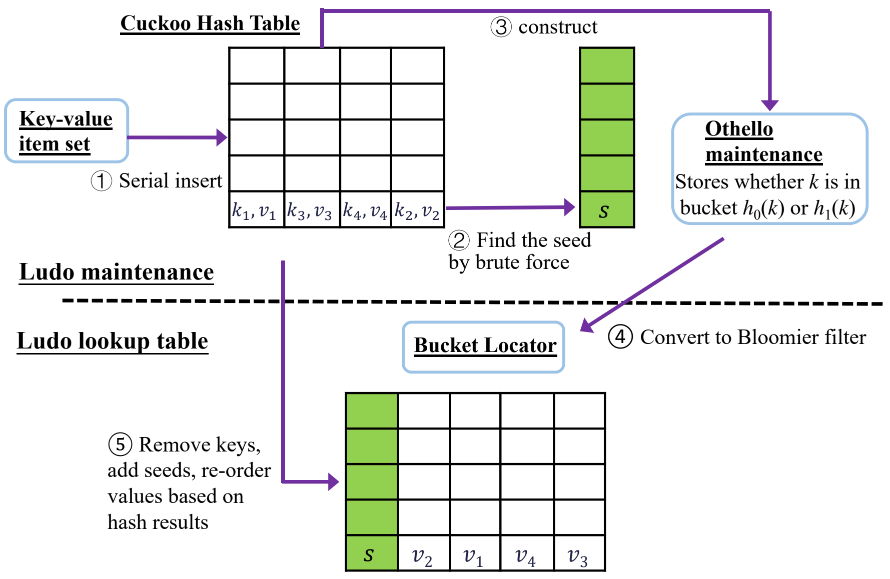
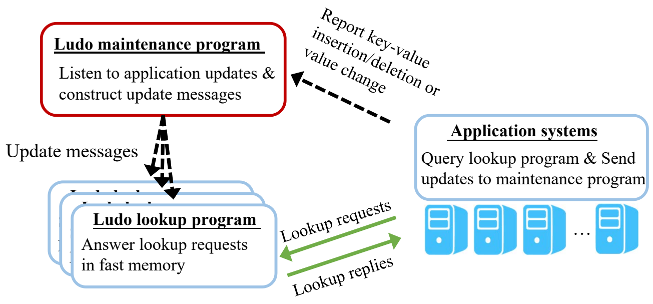


Figure 4.9: Ludo construction algorithm

for all keys.

Step 4. Construct the Othello lookup structure by simply copying the two data arrays from the Othello maintenance structure. Hence, the Othello lookup and maintenance structures give the same lookup result for every input key.

Step 5. Construct a table with the same number of buckets as the source Cuckoo table. For each bucket in the source Cuckoo table (called the source bucket), copy the seed s to the bucket in the same position as the target table (called the target bucket). For each key-value item $\langle k, v\rangle$ in the source bucket, copy v into the $\mathcal{H}_s(k)$-th slot of the target bucket.



Ludo maintenance program
Listen to application updates & construct update messages
Report key-value insertion/deletion or value change
Update messages
Application systems
Query lookup program & Send updates to maintenance program
Ludo lookup program
Answer lookup requests in fast memory
Lookup requests
Lookup replies

Figure 4.10: Ludo hashing system at runtime

4.4.8 Ludo hashing update algorithm

As a part of a practical system, Ludo hashing at runtime consists of two kinds of processes: the Ludo maintenance program holding a Ludo maintenance structure to maintain the full system state, possibly duplicated for robustness, and the Ludo maintenance program running as multiple instances (e.g., multiple lookup servers or routers), as shown in Fig. 4.10. The Ludo maintenance program receives update reports from applications, constructs update messages according to its current state, and sends them to all Ludo lookup programs. Each Ludo lookup program answers the lookup queries from applications and updates its memory according to the messages from the Ludo maintenance program. Similar to other key-value lookup tables, Ludo hashing has three kinds of updates: key-value item insertions, item deletions, and value changes. We discuss the three update algorithms separately.

Item insertion. The Ludo maintenance program takes three steps to construct the update message for an item insertion. 1) It first inserts this key-value item $\langle k, v\rangle$ into the source Cuckoo table and records the *cuckoo path*. The cuckoo path

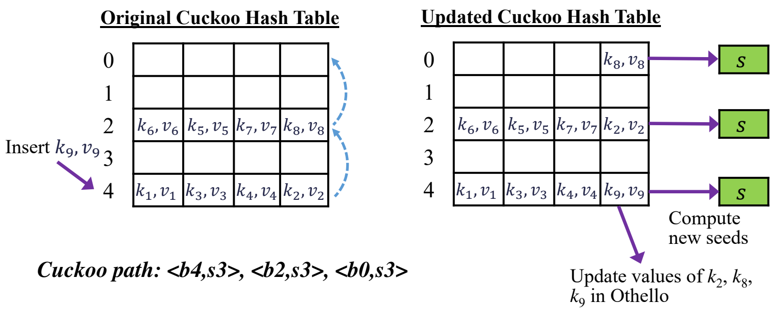

Original Cuckoo Hash Table
Updated Cuckoo Hash Table
Insert k_9, v_9
k_6, v_6
k_5, v_5
k_7, v_7
k_8, v_8
k_1, v_1
k_3, v_3
k_4, v_4
k_2, v_2
k_9, v_9
s
Compute new seeds
Update values of k_2, k_8, k_9 in Othello
Cuckoo path: <b4,s3>, <b2,s3>, <b0,s3>

Figure 4.11: Ludo maintenance program insertion process

of an item insertion is defined as the sequence of the positions of key relocations, where each position is determined by $(bucket\_index, slot\_index)$ [40]. In the example of Fig. 4.11, $\langle k_9, v_9 \rangle$ is inserted into the table. $\langle k_2, v_2 \rangle$ is relocated to from position $(b4, s3)$ to $(b2, s3)$ and $\langle k_8, v_8 \rangle$ is relocated from $(b2, s3)$ to $(b0, s3)$. Hence the cuckoo path is $(b4, s3), (b2, s3), (b0, s3)$. 2) For each relocated key-value item, its position is switched between its alternate buckets $h_0(k)$ and $h_1(k)$. In Fig. 4.11, both k_2 and k_8 have switched between their alternate buckets. The ludo maintenance program updates the corresponding value in the Othello maintenance structure and makes the changes in the Othello lookup structure. 3) For each modified bucket, the Ludo maintenance program finds a new slot locator seed by brute force. The pseudocode is shown in Appendix A.2.1.

When the Ludo maintenance program finishes updating by the above steps, it creates an update message including three fields: *type* tells the update message type (insertion, deletion, or change), *val* is the value of the new item for insertion, and *update_sequence* is a sequence of nodes, representing the updates applied to the Ludo lookup structure. Each node in *update_sequence* corresponds to a position in the cuckoo path and includes the following: the bucket index $bIdx$, slot index $sIdx$, the new seed of this bucket s, the new order of values in the slots of this bucket $vodr$, and the changes made to the Othello lookup structure $Ochg$. The pseudocode of the update steps is shown in Appendix A.2.1.

All associated Ludo lookup programs receive the same update message and follow the update sequence in that message to perform the insertion. Each Ludo lookup program traverses the nodes of the update sequence *reversely* and takes three steps at each node: 1) Copy the bucket indicated in the node to a temporary memory. 2) Write the new seed into the bucket, reorder values according to $vodr$. 3) Atomically write the bucket back to the table and apply the change to the Othello lookup structure. The pseudocode of the update steps is shown in Appendix A.2.1. The compiler barriers and version array are necessary for concurrent reads during updates.

Item deletion. In the Ludo maintenance program, deletions serve for space reclaim for future new items, and a deletion is achieved by deleting the item in the source Cuckoo table and the associated bucket location information in the Othello maintenance structure. There is no change to the Ludo lookup structure. If the number of items is lower than a threshold, e.g., the load factor $< 80\%$, a reconstruction can be triggered on the maintenance program to reduce the size of the lookup structure. During that process, the lookups are still on the existing lookup structure.

Value change. A value change only involves an update to a single slot and does not require any change in the bucket/slot locators. The Ludo maintenance program will perform a lookup in the source Cuckoo table to locate the bucket/slot position of the item, change the corresponding value, and send out a value change message to the lookup structure, specifying the new value and its location in the target table. The Ludo lookup program will perform the value change according to the message.

Consistency under concurrent read/write. We design Ludo hashing as a dynamic key-value lookup table under the single writer multiple reader model. To make the Othello lookup structure work well under concurrency, all modifications to the nodes belonging to the same key should appear atomic to the lookup threads. To allow concurrency in the lookup table, the value reordering should use the reverse order in the update sequence, and sequential writes of a single bucket should be atomic to the lookup threads. We extend the version-based optimistic locking scheme proposed in [40] and [107] for the target Cuckoo table and Othello lookup structure, respectively. Besides, we use the lock striping method proposed in [40] to reduce the size of the version array from the number of buckets to a constant 8192 at a 0.01% false retry rate. The pseudocode is shown in Appendix A.2.1.

Ludo reconstruction. In very rare cases, such as table resizing, the Ludo maintenance program needs to reconstruct the Ludo lookup structure. During the reconstruction time, the data plane still queries the old lookup structure and use the fallback table to guarantee correctness. When reconstruction finishes, the new lookup structure is sent from the maintenance program to the data plane. The update operations on the lookup structures are atomic. The new lookup structure is loaded from the update message, and the old lookup structure is immediately discarded. Since then, the queries will be based on the new lookup structure.

Parallel updates. The update algorithm on the maintenance program can be at some level of parallelism. If two updates do not touch the same bucket, then they can be computed in two threads without violating correctness. The requirement is to have a shared array to store the locks of the buckets. If a bucket is currently in writing, the lock is set to 1, and other threads must wait to visit this bucket. We do not implement the parallel version of updates because the current update speed (>1M operations per second) is sufficiently high.

4.5 Analysis

We summarize the performance analysis of Ludo hashing: 1) The space cost of the Ludo hashing is $3.76 + 1.05l$ bits per item; 2) Each lookup costs 3.02 memory loads on average; 3) Each insertion, deletion, or value change costs $O(1)$ time on average; 4) the communication cost for each update is $O(1)$ on average. The following presents the details.

4.5.1 Space cost of Ludo lookup structure

A Ludo lookup structure consists of three parts: the Bloomier filter for the bucket locator, the lookup table storing values and seeds, and a small table for the overflow seeds. The Bloomier filter costs 2.33 bits per key. The seeds cost 5 bits per bucket, i.e., 1.25 bits per key. The overflow table contains 1.2% of the seeds statistically, and each entry in the overflow table costs $29 + 8 = 37$ bits. Since the load factor of the Ludo lookup structure is 95%, it costs $1.05l$ bits per item, where l is the length of each value in bits. In total, the average memory cost per key-value item is: $2.33+5\times 1.05/4+37\times 1.05\times 0.012/4+1.05l = 3.76+1.05l$ bits. The space cost of the fallback table is $O(n_f)$, where n_f is the number of fallback keys and $n_f \to 0$ based on the insertion correctness analysis below. Also, our experiments never find a single fallback key. When the lookup structured is updated, the load factor may be set to an application-specific threshold (such as 94%). Hence the space cost may increase to $3.78+1.06l$.

4.5.2 Lookup overhead

A key-value lookup in Ludo lookup structures always requires 3 memory loads: two for the Bloomier filter, and one to fetch the bucket holding the value. If the seed overflows (with probability 1.2%), another 1 or 2 random loads are required in the overflow table to get the seed. Hence we get the average number of memory loads $3+0.012\times(1\times 0.71+2\times 0.29)=3.016$.

4.5.3 Insertion correctness

From existing theoretical results of random graphs presented by Cain et al. [31] and independently Fernholz and Ramachandran [45], it has been proved that all n keys can be inserted into a (2,4)-Cuckoo table asymptotically almost surely (a.a.s.) such that each bucket has at most 4 keys if the load factor $< 98.03\%$, assuming uniform hashing and $n \to \infty$. This result has been confirmed by later studies [48, 46, 61, 98]. A detailed explanation can be found in the Appendix A.2.2. In practice, our design sets the load factor threshold to 95% to avoid hitting the tight threshold. In fact, we have not observed a single failure case among over 20 billion insertions during our tests.

When the load factor $< 95\%$, the insertions are unlikely to fail from the above results. When the Ludo maintenance program detects the current load factor reaches 94%, it will start to build a new Cuckoo table with a higher capacity. The insertion failures (if any) will be stored in the fallback table. This design guarantees correctness via these properties: 1) the runtime load factor will not be higher than 95% in most time; 2) even if the load factor temporarily exceeds 95% while the rebuild of Ludo with higher capacity has not finished, most insertions are still successful as the theoretical threshold is 98%; 3) even if there is an insertion failure, the fallback table is able to store it and guarantees correctness of lookups.

4.5.4 Update overhead

Item insertion. The time complexity of each insertion to Ludo includes three parts: 1) the time to add the item to Othello; 2) the number of nodes in the update sequence of each Cuckoo insertion, and 3) the time of updating the bucket of each node. We show the time of each insertion to Ludo is amortized $O(1)$ and independent of n based on the facts that all these three parts are either $O(1)$ or amortized $O(1)$. Inserting an item to Othello is proved to be amortized $O(1)$ [107]. From the theoretical results in [68], for a $(2,k)$-Cuckoo with load factor $1/(1+\epsilon)$ and $k \geq 16(\ln(1/\epsilon))$, each insertion costs amortized constant time $((1/\epsilon)^{O(\log\log(1/\epsilon))})$ by breadth-first search [68, 98]. Our design uses $k = 4$, which is less than $16(\ln(1/\epsilon))$. There is no proof of constant-time insertion for this setting. In our experiments, all insertions finish within 5 levels of breadth-first search. For each node in the update sequence, the update includes re-compute a seed (up to 31 attempts) and re-ordering the values (up to 4). It costs constant time for each node. We list the lengths of the update message fields. *type*: 1 bit; For each node in the update sequence, *bIdx*: 30 bits; *sIdx*: 2 bits; *seed*: 8 bits; *vorder*: 2 bits for each slot and 8 bits in total; *Bchg* contains the indices of the influenced nodes in the Bloomier filter, 32 bits for each index.

Each item deletion or value change costs $O(1)$ time and communication cost.

We discussed the average case above. In the worst case (very rare), an update may cause a reconstruction of Othello, but it only happens with probability $O(1/n)$ as proved in [107]. The Cuckoo table will not experience reconstructions when the load factor is no more than 95%, as shown above.

4.6 Implementation and evaluations

4.6.1 Evaluation methodology

In this section, we conduct two types of performance evaluation of Ludo hashing: 1) Evaluation of the in-memory lookup tables on a commodity workstation with two Intel E5-2660 v3 10-core CPUs at 2.60GHz, with 160GB 2133MHz DDR4 memory and 25MB LLC; 2) Case study of Ludo hashing on two real network systems, namely distributed content storage and packet forwarding.

We implement the Ludo maintenance structure and Ludo lookup structure prototypes in 3272 lines of C++ code. We also make use of the open source implementation of Cuckoo Hashing (*presized_cuckoo_map* in the Tensorflow repository [11]) and Othello Hashing (its authors' implementation [4]), with several major modifications to implement bucket/slot locator, update, and concurrent reading/writing. The buckets of Ludo lookup structure are stored as an array of 64-bit integers by carefully applying a series of bit-wise operations, such that there is no single bit waste on storing the buckets. The source code of Ludo hashing is available for results reproducibility [3].

We identify the following metrics to be evaluated:

1. **Memory cost**, the most important metric to characterize the space efficiency.

2. **Speed of update** to characterize the update time.

3. **Lookup throughput** for single thread, multiple threads, and with concurrent reading/writing.

4. **Construction time** of the lookup engine.

Each data point shown in the figures is the average of 10 independent experimental runs. We also use the error bars to show the standard deviation among the 10 results. For lookup throughput evaluations, the request workloads are in two types: in the uniform distribution and Zipfian distribution. For the uniform distribution, all items are requested with an equal probability. For the Zipfian distribution, items are requested with biased probabilities, which better simulates the workload in most practical systems. We set the Zipfian parameter to be 1.

We compare Ludo hashing with the following dynamic lookup solutions: (2,4)-Cuckoo [74, 40], partial key Cuckoo [64, 87], Othello Hashing [107], and SetSep [42, 110].

We implement partial key Cuckoo based on the Tensorflow repository [11], with several major modifications to support fingerprint collision resolution. We implement SetSep and made several extensions to allow some level of updates of SetSep after construction – but still, reconstructions are frequently needed. We use Google FarmHash [2] as the hash function for all experiments.

4.6.2 Evaluation of in-memory lookup engines

We denote the number of key-value items as n, the sizes of each value, key, and digest as l, L, and L', respectively, all in bits.

Memory cost. Fig. 4.12 shows the memory cost breakdown of Ludo lookup structure, SetSep, Othello Hashing lookup structure, Cuckoo hashing, and partial key Cuckoo hashing, where $n = 1$B, $L = 100$, and $L' = 30$. We set l as 10 and 20. Clearly, Ludo hashing needs the least memory cost among all designs for both $l = 10$ and 20. By comparing the breakdown parts of each design, we find that Ludo uses a similar space to store the values, which seems unavoidable for every key-value lookup table. Note that Othello embeds the values in the two arrays A and B. Ludo saves much space cost by reducing the key storage while maintaining a low amplification on value storage. Despite being difficult to update, SetSep costs more space than Ludo hashing, especially for large l.

From the analytical comparison in Fig. 1.2, Ludo always costs the least memory when $l > 3$. We then compare the actual memory cost of the in-memory lookup tables in three practical setups. 1) For the application of indexing distributed contents, we set

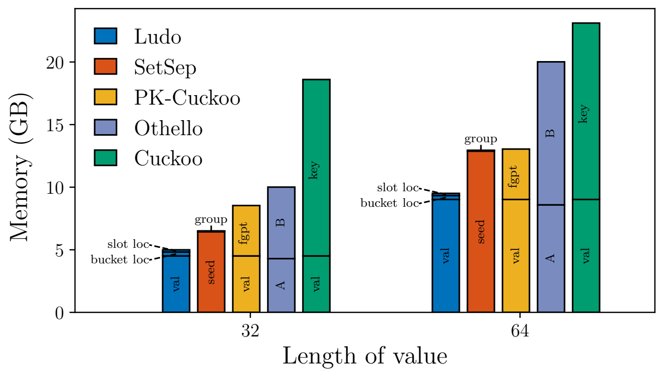
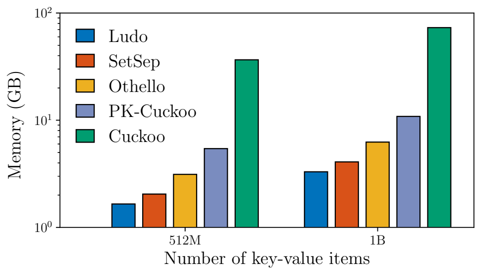


Figure 4.12: Memory cost for different value lengths (1B keys)

$l = 20, L = 500, L' = 60$, assuming there are 1M content storage nodes. We set n to be 512M and 1B and show the results in Fig. 4.13. Ludo only requires 3.3GB for 1B items, while other designs need at least 6.3GB. Here Ludo saves almost 50% memory. 2) For the application of network FIBs, we set $l = 8, L = 48, L' = 30$, assuming a switch has 256 ports and MAC addresses are used. The results are shown in Fig. 4.14. It is known that a commodity switch has < 100MB SRAM [67], and Ludo only needs 50.5MB for 32M addresses. 3) For the application of indexing key-value storage, we set $l = 40, L = 200, L' = 60$ and show the results in Fig. 4.15. Ludo only uses 6.1GB memory to support 1B items, while other designs need > 12GB.

Dynamic update. We evaluate the update throughput of Ludo hashing, Othello Hashing, and SetSep, which characterizes the maximum number of updates a table can support in the unit of millions of operations per second (Mops). All experiments are performed in a single thread, with equal numbers of insertions, deletions, and changes. We set $L = 32$ and $l = 20$, change the table size, and show the results in Fig. 4.16 where SetSep only performs the updates that do not cause reconstruction. Each update event



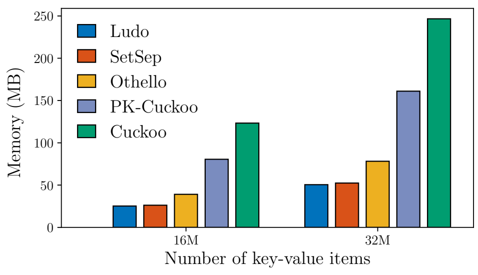
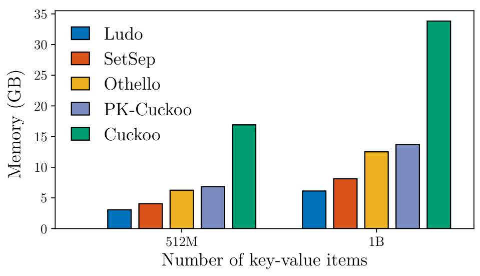


Figure 4.13: Memory cost for indexing distributed contents

Figure 4.14: Memory cost for FIBs



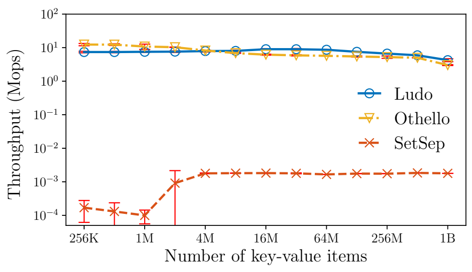


Figure 4.15: Memory cost for indexing key-value storage

Figure 4.16: Throughput (speed) of updates

may be an insertion, deletion, or value change, with equal probability. The results show that Ludo allows > 5Mops updates, which is sufficient for most applications. Othello shows comparable performance with Ludo, while SetSep performs > 1000x worse than the other two even if we only consider the updates that do not cause reconstruction. As shown in the results below, each reconstruction of SetSep may take hundreds of seconds to >5 hours.

Single-thread lookup throughput. We compare the single-thread lookup throughput of Ludo, Othello, partial key Cuckoo, and SetSep, in Zipfian (Fig. 4.17)

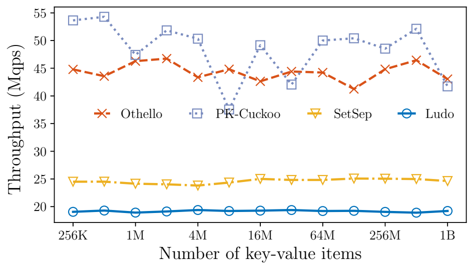


Figure 4.17: Single-thread throughput for Zipfian

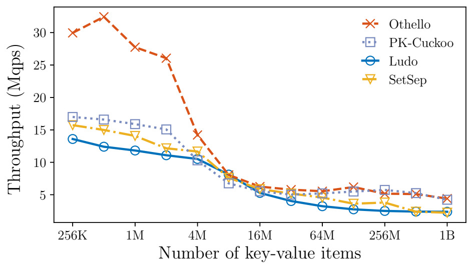
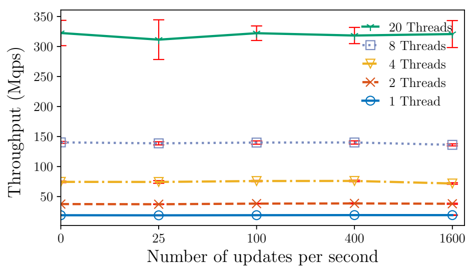


Figure 4.18: Single-thread throughput for uniform

and uniform (Fig. 4.18) workload, respectively. We set $l = 20$ and vary n, and the throughputs are in the unit of million queries per second (Mqps).

The throughput under uniform queries decreases with the growth of table size because the memory is randomly accessed, and a larger table incurs a higher cache miss rate. The throughput under Zipfian distribution is less degraded by the table size because the L3 cache satisfies most queries. Othello/Bloomier shows the highest lookup throughput. Ludo hashing is slower because, for a single lookup, it requires 1 more hash function calculation and 1 more memory load. However, it still satisfies > 5M queries per second when $n \leqslant$ 16M and > 3M when $n =$ 1B. The throughput satisfies most applications and unlikely to become the system bottleneck.

Throughput under updates. We wonder whether concurrent writing/reading would affect the performance. Fig. 4.19 shows the lookup throughput of Ludo under concurrent updates (writing) by varying the update frequency, where $L = 64$, $l = 20$, and $n =$ 16M. Our observation is that there is no noticeable throughput degradation



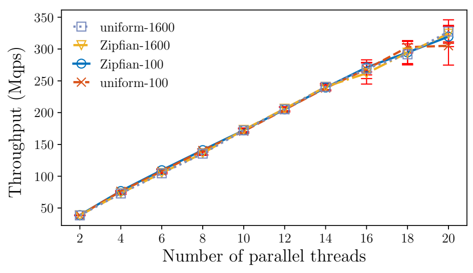


Figure 4.19: Lookup throughput under updates

Figure 4.20: Lookup throughput scales with # of threads

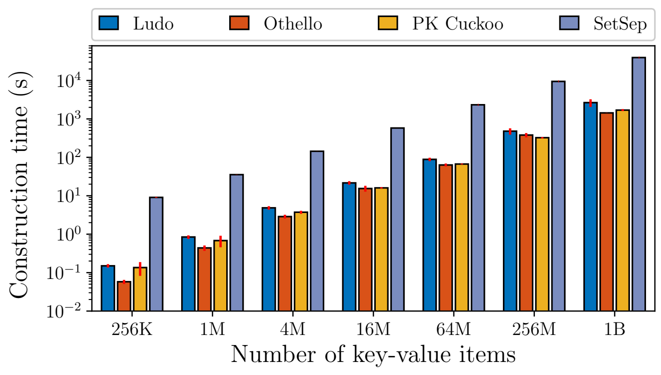


Figure 4.21: Construction time

when the update frequency grows to up to 1.6K updates per second. Since 1.6K updates per second are sufficient for most dynamic applications, we may conclude that the lookup throughput is stable under concurrent writing.

Multi-thread throughput. We also show the results of multi-thread lookup throughput in Fig. 4.20, with up to 20 threads on a single machine and concurrent updates (100 and 1600 times per second). We find that the throughput scales linearly with the multi-thread. It achieves $>$ 300Mqps with 20 threads for $n = 1$B.

Construction time. We also examine the construction time of the lookup engines. Fig. 4.21 shows the construction time of different designs by varying n, for $L = 64$, and $l = 20$. SetSep is >10x slower than other tables and takes >5.5 hours to construct for 1B keys. All other tables have similar construction time. For 1B items, Ludo hashing can be constructed in 30 minutes.

4.6.3 Case studies of real systems

We study the practical system performance with Ludo hashing for two applications. **All experiments in this subsection perform real query packet receiving and forwarding.**

4.6.4 Case 1: indexing distributed contents

In this system, a large number of data contents are stored among the distributed storage nodes. There is an index node that accepts the queries of contents and forwards them to the correct storage nodes. The index node can be easily replicated to avoid the single point of failure. This model may be applied to many practical systems such as distributed data storage in a data center [23], CDNs [65], or edge computing [89]. In our experiments, the requested keys are uniformly sampled from the *std::string* representation of the content IDs (45 bytes).

Implementation details. We run the experiments in CloudLab [1], a research infrastructure to host experiments for real networks and systems. We implement Ludo hashing, Bloom filter based lookup table (Summary Cache [43]), partial key Cuckoo hashing, and Othello Hashing to serve as the content lookup engine. We use two nodes 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. They are denoted by Node 1 and Node 2 in the following presentation. Each node has two Intel E5-2660 v3 10-core CPUs at 2.60GHz. The Ethernet connection between the two nodes is 2x10Gbps. The network between the two nodes provides full bandwidth. Logically, Node 1 works as the index node, and Node 2 works as all storage nodes in the system. The clients generate queries from the content IDs with Zipfian and uniform distributions.

Throughput of query processing and forwarding. We evaluate the query processing and forwarding throughput of Ludo hashing, Bloom filters, partial key Cuckoo, and Othello in the distributed content storage system. The measurements are in million queries per second (Mqps). We vary the number of contents from 16K to 16M. Figures 4.22 to 4.25 show the throughput versus number of items, in single and two threads, with Zipfian and uniform workload, respectively. Ludo hashing provides the highest throughput as the index among the four methods. The reason is that the bucket locator of Ludo hashing is compact enough to fit into the L3 cache so that it is likely to have only one load from the main memory for the table bucket access. Other solutions may have two main memory loads. Another interesting observation is that the capacity of querying processing and forwarding is bounded by 7 Mqps, which is smaller than the network bandwidth. The throughput does not grow significantly when we add more threads, which infers computation is not the bottleneck. Hence we consider the throughput is bounded by the bus bandwidth between CPU and memory.

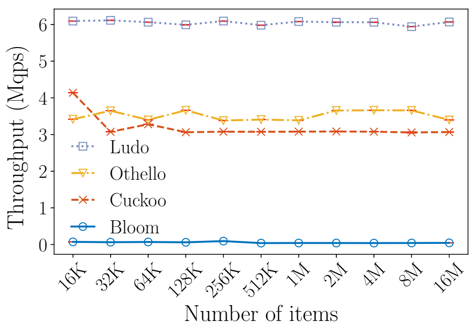


Figure 4.22: Throughput of querying contents with Zipfian workload (single thread)

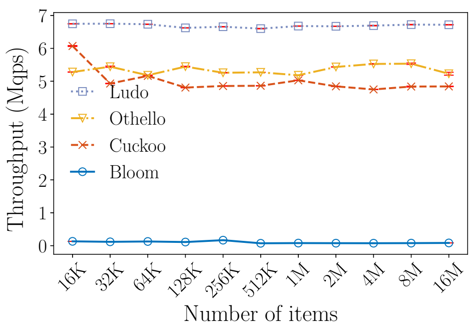


Figure 4.23: Throughput of querying contents with Zipfian workload (two threads)

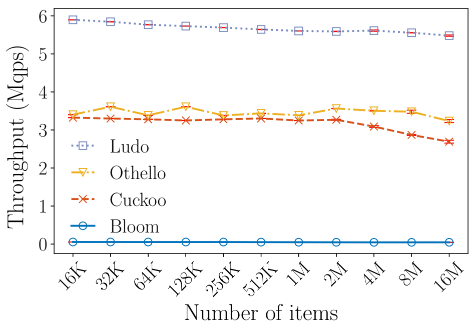


Figure 4.24: Throughput of querying contents with uniform workload (single thread)

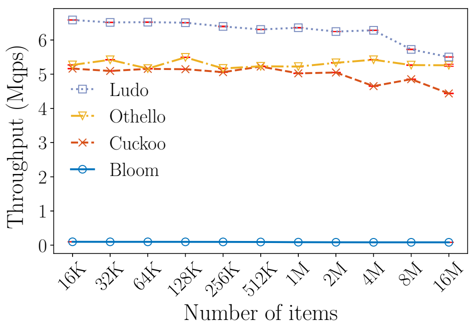


Figure 4.25: Throughput of querying contents with uniform workload (two threads)

4.6.5 Case 2: forwarding information bases (FIBs)

A modern data center network includes a large number of physical servers [56, 50, 80]. Each server is identified by its network address (e.g., its MAC address). An interconnection of switches connects the servers. Each switch has multiple ports connecting neighboring switches and servers. A switch forwards the packet to a neighbor based on FIB lookups using the packet address. Many modern networks are variants of this model [56, 50, 80]. For software defined networks [69], the flow ID may be a combination of source/destination IPs, MACs, and other header fields. The forwarding may be on a per-flow basis rather than a per-destination basis. LTE backhaul networks and core networks can also be regarded as an instance of this network model, especially for the down streams from the Internet to mobile phones, where the destination addresses are Tunnel End Point Identifiers (TEIDs) of mobiles [110].

Implementation details. In the CloudLab prototype, we implement the FIBs as software switches [111, 107] that are running on the end hosts. We implement the FIBs using Ludo hashing, Bloom filter based method (Buffalo [104]), partial key Cuckoo hashing [111], and Othello Hashing [107]. For each FIB implementation, we make several major modifications to support Dijkstra routing. The prototypes work with Intel Data Plane Development Kit (DPDK) [5] to support packet forwarding using end hosts. DPDK is a series of libraries for fast user-space packet processing [5] and is useful for bypassing the complex networking stack in the Linux kernel, and it has utility functions for huge-page memory allocation and lockless FIFO, etc. We modify the code of the key-value lookup tables and link them with DPDK libraries. The query keys are in four types: 32-bit IPv4 addresses, 48-bit MAC addresses, 128-bit IPv6 addresses, and 104-bit 5-tuples. We still use the two nodes in CloudLab (denoted by Nodes 1 and 2) for this prototype. The Ethernet connection between the two nodes is 2x10Gbps. The switches between the two nodes support OpenFlow [69] and provide full bandwidth. Logically, Node 1 works as a switch in the network, and Node 2 works as the neighboring switches and end hosts in the network.

Node 2 uses the DPDK official packet generator Pktgen-DPDK [6] to generate random packets and sends them to Node 1. The packets sent from Node 2 carry the destination addresses with Zipfian or uniform distributions. Each FIB prototype is deployed on Node 1 and forwards each packet back to Node 2 after determining the outbound link 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 addresses, will be received by Node 2. Node 2 then records the receiving bandwidth as the throughput of the whole system. The maximum network bandwidth is 28.40 million packets per second (Mpps).

Packet forwarding throughput. Figures 4.26 to 4.29 show the packet forwarding throughput of the four solutions, by vary the number of addresses stored in the FIB, with Zipfian and uniform distributions, for single thread and two threads, respectively. While Othello Hashing performs the best on a single thread, two threads of Ludo hashing, partial key Cuckoo hashing, and Othello Hashing are sufficient to fill the full network bandwidth (called line rate) for a 16M FIB. For all cases, FIBs with

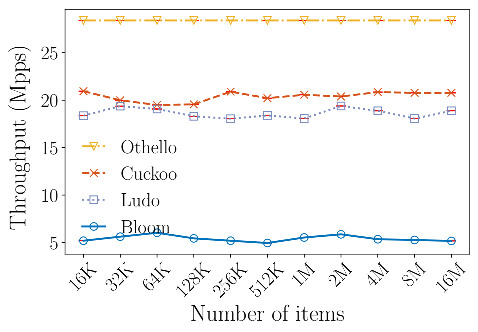


Figure 4.26: FIB throughput with Zipfian workload (single thread)

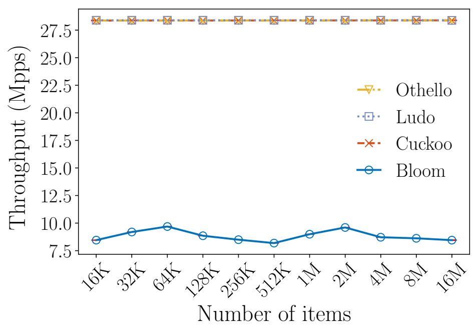


Figure 4.27: FIB throughput with Zipfian workload (two threads)

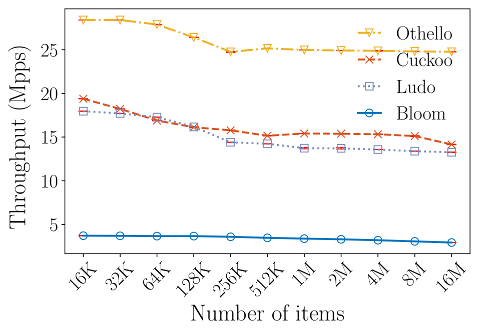


Figure 4.28: FIB throughput with uniform workload (single thread)

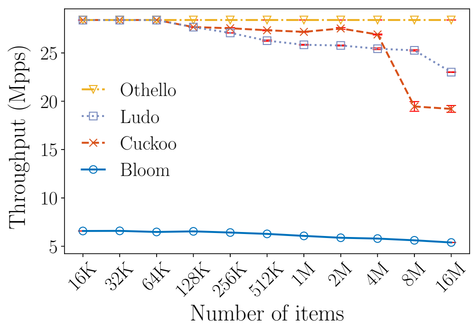
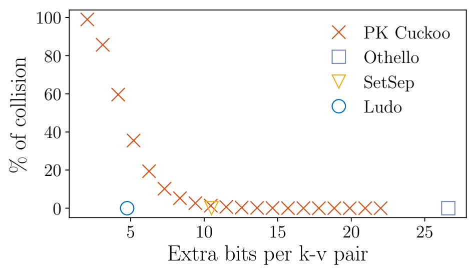


Figure 4.29: FIB throughput with uniform workload (two threads)

Ludo hashing, Othello Hashing, and partial key Cuckoo hashing performs >2x higher throughput than Bloom filters.



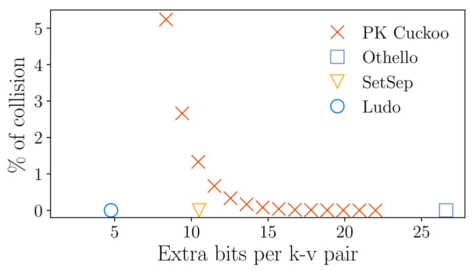


Figure 4.30: Memory cost and collision rate

4.6.6 Summary of evaluation

Memory footprint. Ludo hashing is the most compact among all dynamic in-memory lookup tables under all configurations.

Lookup throughput. Ludo lookup structure achieves 5 to 20 Mqps single-thread throughput for up to 1B items. The throughput scales linearly with the number of threads and can achieve 65Mqps on one node.

Runtime update. Ludo lookup structure performs > 6M updates per second. The throughput of Ludo lookup structure is stable with concurrent updates.

Construction time. Ludo hashing can be constructed for 1B items in 10 minutes.

Performance in real systems. Ludo hashing provides higher throughput than other methods in the content lookup system. In the packet forwarding system, Ludo hashing can easily achieve maximum network bandwidth with two threads.

4.7 Discussion

Partial-key Cuckoo. One may consider setting short digests in partial-key Cuckoo [64] is a straightforward solution. However, short digests cannot be used because the key collision rate grows. Assuming values are l-bit long and $l = 20$, we change the key digest bit length L' from 1 to 20 for partial key Cuckoo and observe the relation between the extra memory cost and key collision rate. The extra memory cost is defined as the overall memory cost of the lookup data structure minus nl, where n is the number of keys. We insert 1M random MAC addresses into different partial key Cuckoos, and the results are shown in Fig. 4.30.The right figure zooms in and shows the results near 1% of key collision. If we configure the PK Cuckoo to take no more than the memory of Ludo, $> 40\%$ keys will be mapped to more than one value. If we control the collision rate under 0.1%, the PK Cuckoo takes > 3x extra memory than Ludo.

Alien keys. Let K be the set of the keys of all items. An alien key (k_α) is defined as a key that was never inserted into the item set, i.e., $k_\alpha \notin K$. The lookup of an alien key may result in an arbitrary value by a perfect hash table, and we denote this as the 'alien key problem'. The alien key problem is not unique for Ludo. It exists for all perfect hashing based designs that do not store keys, including SetSep [42], Bloomier filters [28], and Othello [107]. There is a simple trade-off: either store the keys with several times higher memory cost or accept the alien key problem and try to limit its impact. However, for any key $k \in K$, the lookup by Ludo hashing will always be correct. Hence there is no false lookup result.

Most applications in the context of this work are not sensitive to alien keys, namely the distributed content index, network forwarding, and storage index. For a distributed content index, querying an alien key will make the index forward the request to an arbitrary storage node. The storage node will then find that no data in the node match this key. Hence it simply notifies the client of a 'not exist' message. For a network forwarding device, a packet with an alien address will be forwarded to an arbitrary port. Note that every packet will carry the time-to-live (TTL) field that will decrease by 1 after each forwarding action. Hence a packet will either be dropped when the TTL becomes 0 or dropped at a destination that does not match the address. Also, most networks will have firewalls that can filter all packets with alien addresses. In the above situations, an alien key has a limited negative impact.

Alien keys will become a problem for applications that need to filter keys such as firewalls. Hence none of the perfect hashing methods can be used for firewalls. For applications that really need to filter alien keys, a filter function can be added to the lookup table. Ludo hashing can be perfectly combined with a Cuckoo filter [41, 92] to have a better trade-off between false positives and memory compared to Bloom filters. Other methods such as Othello and SetSep will need either extra memory or lookup time to work with a filter. This topic is beyond the scope of this work, and we skip the details due to space limit.

Chapter 5

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

5.1 Overview

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. Compared to large clouds, the emerging edge data centers bring more LB design challenges, including resource efficiency, consistency of multi-connection state, and weighted load balancing. However, current stateful software LBs face a dilemma: storing all states in the LB incurs high resource cost while using digests causes false hits and table explosion due to collisions. 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 and consistent under network changes and incurs low update cost. Unique features of Concury include 1) packet consistency with extremely low update frequency; and 2) low memory cost without inconsistency, resulting in higher throughput than other stateful LB algorithms under network dynamics. 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 – the highest performance to our knowledge.

5.2 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 [77, 47, 96, 37, 10]. Traditional hardware load balancers are expensive and not flexible. Hence, many large cloud services choose to use software load balancers [77, 47, 37, 10, 19]. In addition, LBs are also important for edge data centers [91, 89, 108, 25], which allow heterogeneous devices on the path to the remote cloud to offer storage and computing resources.

LB Algorithm	Lookup (Mpps)	Memory (MB)	Weighted LB	False hits	Packet type	Extra hardware	Update interrupt
ECMP + hash table (Ananta [77])	low	high	unclear	No	any type	No	frequent
Hash table w/ digest (Maglev [37])	14.63	18.63	Yes	exist	TCP only	No	frequent
Multi HTs w/ digest (SilkRoad [67])	16.11	4.36	No	exist	TCP only	ASIC	frequent
Concury (this work)	66.28	3.84	Yes	No	any type	No	infrequent

Table 5.1: 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 § 5.6.

Stateful load balancers. Ananta [77] 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 a very slow packet processing speed, as shown in [47]. Duet [47] makes use of forwarding and ECMP tables on commodity switches to store VIP-DIP mappings. Under frequent DIP pool changes, Duet may not be able to maintain PCC [67]. Maglev [37] 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 [67] 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 [74]. Bonomi *et al.* proposed to use Approximate Concurrent State Machines (ACSMs) to maintain dynamic network states [29], but this method cannot be used for LBs. We compare Concury with existing stateful LBs in Table 5.1, where the experimental values are based on the DIP-V 16M-state network in 5.6.2.

Stateless load balancers. Beamer [73] and Faild [25] are recently proposed stateless LBs. Their forwarding logics do not store connection states 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 stateful LB design and leave the choice between stateful and stateless LBs to network operators.

5.3 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.

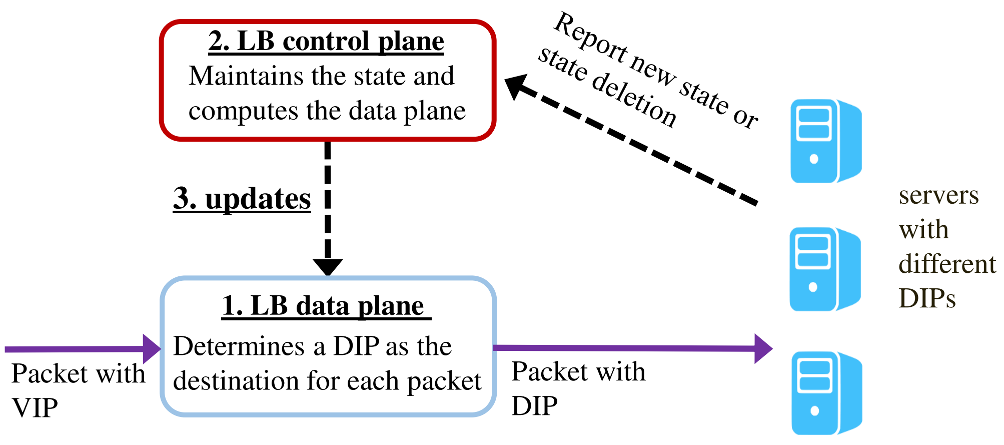

2. LB control plane
Maintains the state and computes the data plane
Report new state or state deletion
3. updates
1. LB data plane
Determines a DIP as the destination for each packet
Packet with VIP
Packet with DIP
servers with different DIPs

Figure 5.1: General model of a stateful LB

Achieving all requirements of an LB stated in § 5.1 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. An example of consistency violation by static hashing is shown in Appendix A.3.1.

Recent stateful LB designs [37, 67] need to store connection states 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. 5.1), 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 [37, 67]. 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.

Although Concury needs the servers to send state notifications, 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 [47].

5.4 Design of Concury

5.4.1 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 § 5.3, 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 [47]. 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 § 5.3.

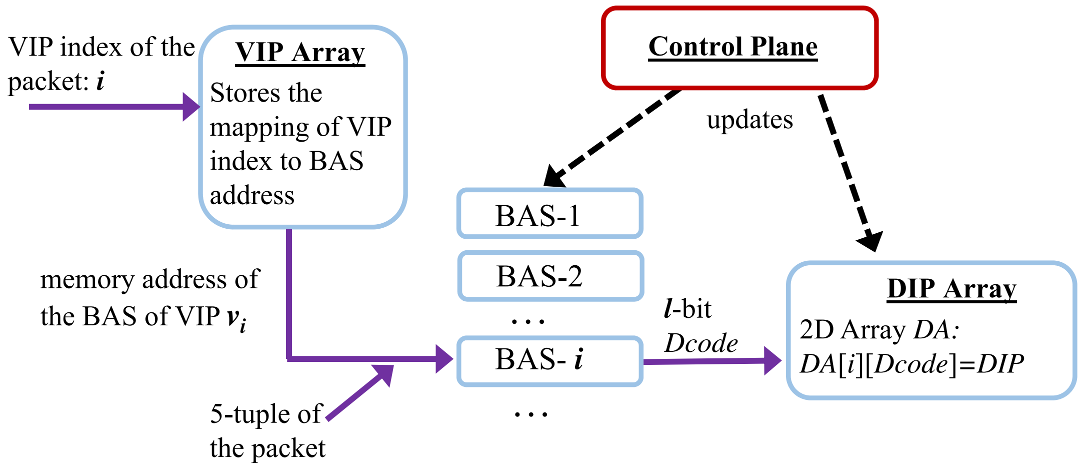

VIP index of the packet: *i*
VIP Array
Stores the mapping of VIP index to BAS address
memory address of the BAS of VIP v_i
5-tuple of the packet
Control Plane
updates
BAS-1
BAS-2
…
BAS- *i*
…
l-bit *Dcode*
DIP Array
2D Array *DA:* *DA*[*i*][*Dcode*]=*DIP*

Figure 5.2: Workflow of Concury data plane

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 include: 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.

5.4.2 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 § 2, 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 [37] to avoid the cross-VIP problem. However, it still cannot resolve the digest-deletion problem stated in § 5.2. Concury is unique because it can deal with both types of problems.

The workflow of Concury data plane is shown in Fig. 5.2, which includes three main steps. We show the pseudocode in Appendix A.3.1. 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 a DIP with one more static table lookup.

Data plane complexity analysis and comparison. Detailed analysis and comparison are presented in Appendix A.3.3. Here we present the results.

1) Time cost. Concury-DP is very simple and fast. Each lookup is in $O(1)$,

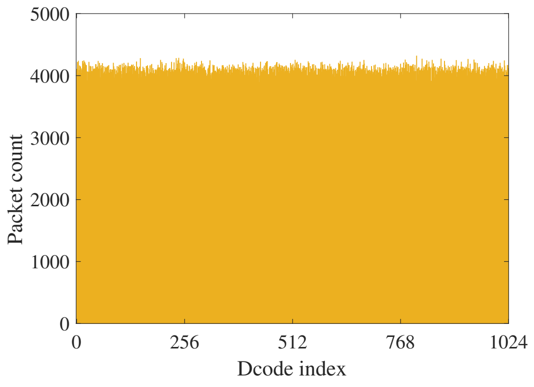


Figure 5.3: Stateless packet distribution by Dcode

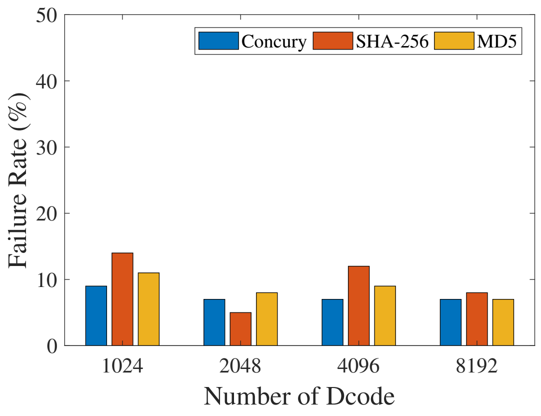


Figure 5.4: Chi-squared test

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 [37, 67], 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 $2.33 l_d n + 64m + 2^{l_d} l_v m + 48 \cdot 2^{l_v}$ bits, which is much smaller than that of Cuckoo+digest in practical setups.

5.4.3 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

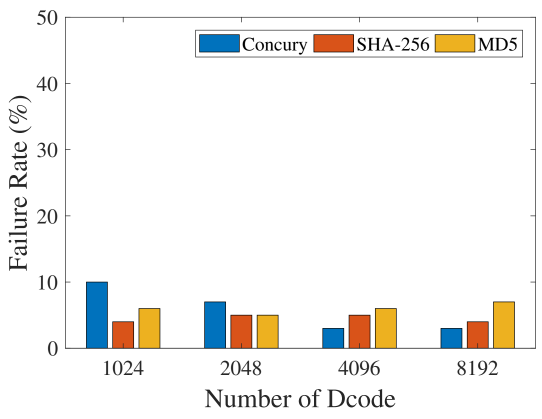


Figure 5.5: Kolmogorov-Smirnov test

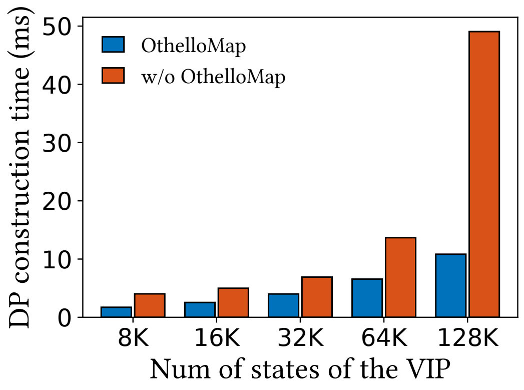


Figure 5.6: Response time of Concury-DP construction

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 an equal probability of being 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. 2.3, 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.

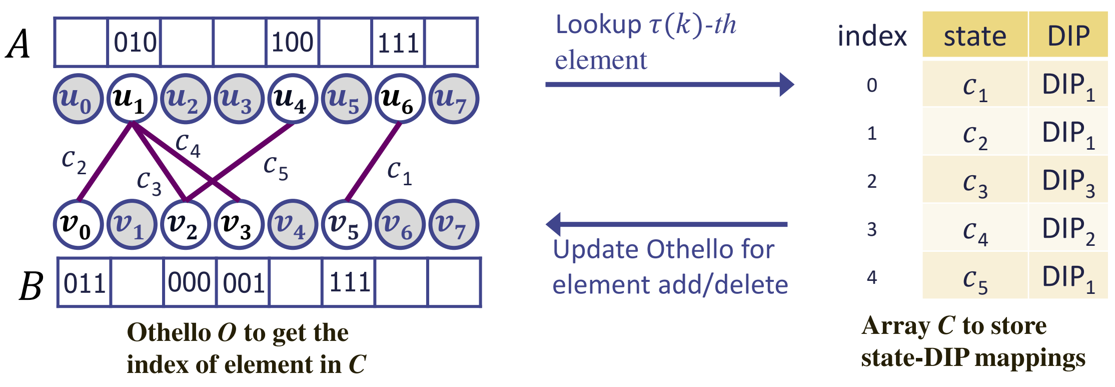
We use empirical results to validate this uniformity. Fig. 5.3 shows one typical example. We let the value length $l = 10$. Hence, there are 1024 possible Dcodes. We enumerate all possible combinations of indexes of A and B and compute the resulting Dcodes. The hash function used in Concury is CRC32. 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. 5.4 and 5.5, 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 § 5.6.6.

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 § 5.4.5.

5.4.4 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 needed, the new BAS is constructed from the set. Our *innovative idea* is to design a new data structure called the OthelloMap



A
010
100
111
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7
c_2
c_4
c_3
c_5
c_1
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
B
011
000
001
111
Othello O to get the index of element in C
Lookup $\tau(k)$-*th* element
Update Othello for element add/delete
Array C to store state-DIP mappings

index	state	DIP
0	c_1	DIP_1
1	c_2	DIP_1
2	c_3	DIP_3
3	c_4	DIP_2
4	c_5	DIP_1

Figure 5.7: Lookup/update of an 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. An illustration is shown in Appendix A.3.4.

Components of an OthelloMap. As shown in Fig. 5.7, 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. 5.6. *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.

5.4.5 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 § 5.4.3, 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.

The pseudocode of Concury-DP updating is in Appendix A.3.2. 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 that Concury updates happen infrequently (once per DIP change) and only apply to the part of data plane structures of one VIP.

5.5 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 $\tau(k)$ equals to *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 5.4.2, 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 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 a 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.

5.6 Implementation and Evaluation

5.6.1 Evaluation methodology

We conduct three types of evaluations: 1) algorithm micro-benchmark; 2) Concury prototype using DPDK [5] deployed in two real networks (100GbE lab network and CloudLab [1]), and 3) a P4 prototype running on Mininet [17]. **Our code is publicly available with an anonymous link [22]. The results can be reproduced.** The purpose of the algorithm micro-benchmark is to compare the algorithms of Concury

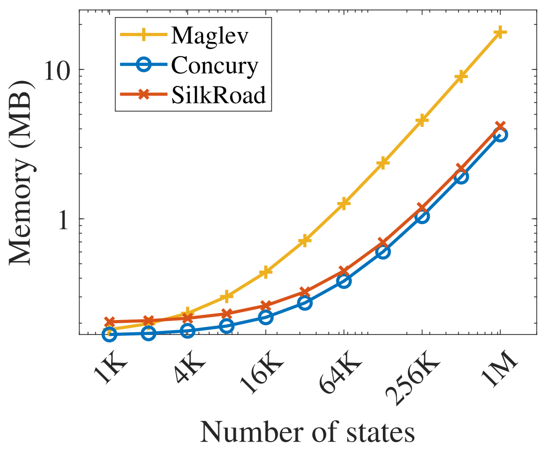


Figure 5.8: Memory cost for DIP-E and Small network

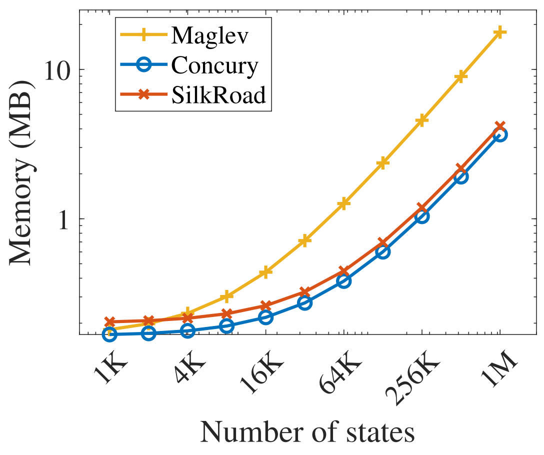


Figure 5.9: Memory cost for DIP-V and Small network

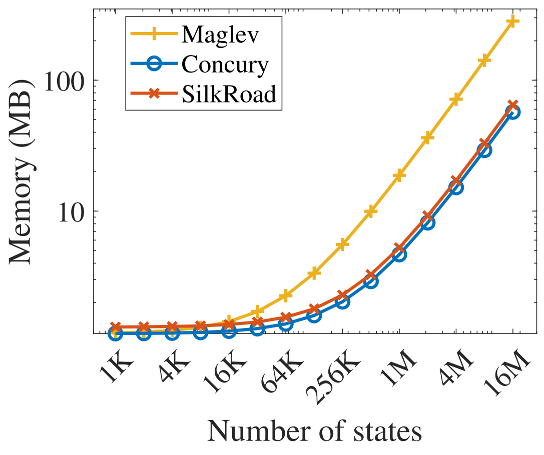


Figure 5.10: Memory cost for DIP-E and Large network

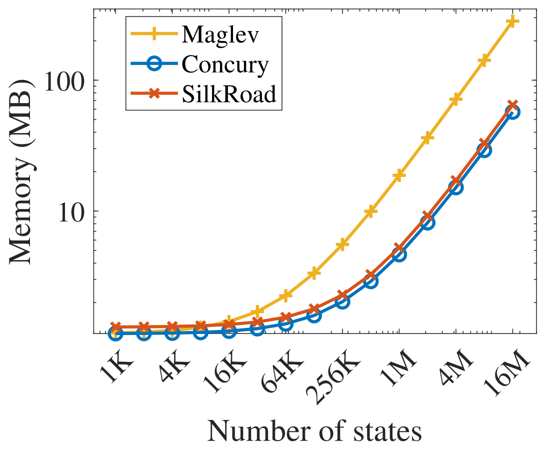


Figure 5.11: Memory cost for DIP-V and Large network

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 [37]; and 2) Multi hash tables with digest, used in SilkRoad [67]. Note SilkRoad was designed for special hardware, i.e., programmable switch ASICs with $>$ 50MB memory. Hence, the performance shown in [67] 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 rebuild identical system prototypes of Maglev and SilkRoad as some of their technique details are not fully presented [37, 67]. In addition, we also separate the hash table of Maglev on 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 [73, 25], such as Beamer [73]. 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 [8] 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

$\texttt{seed}$ be a 32-bit integer. We let $h_a(k) = H(k, \texttt{seed}_a)$ and $h_b(k) = H(k, \texttt{seed}_b)$. Thus, h_a and h_b are uniquely determined by $\texttt{seed}_a$ and $\texttt{seed}_b$, respectively.

We use the **real traffic trace** from the Facebook data center networks [14] 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 [67], 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 thousand to 10M.

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

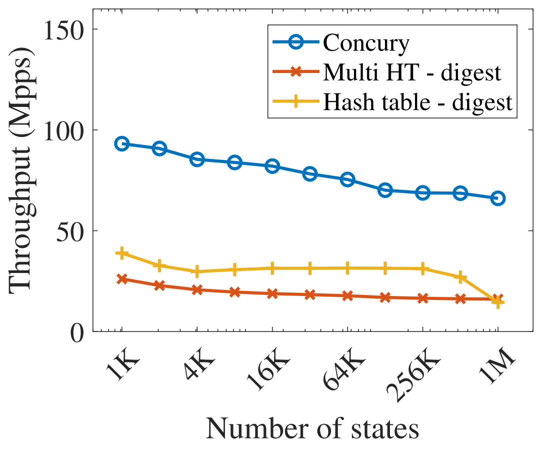


Figure 5.12: Throughput for DIP-E and Small network

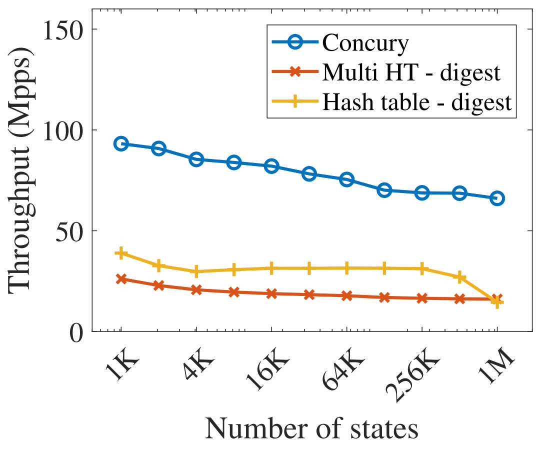


Figure 5.13: Throughput for DIP-V and Small network

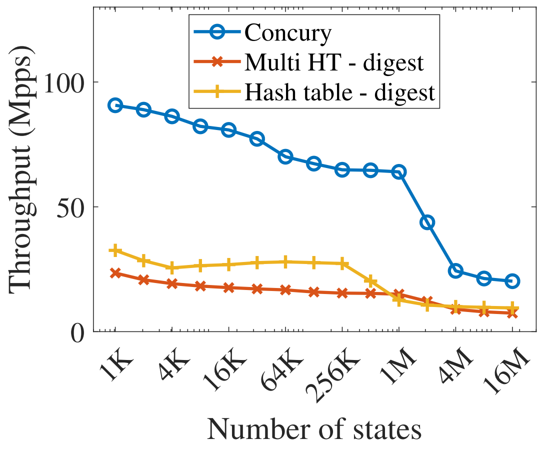


Figure 5.14: Throughput for DIP-E and Large network

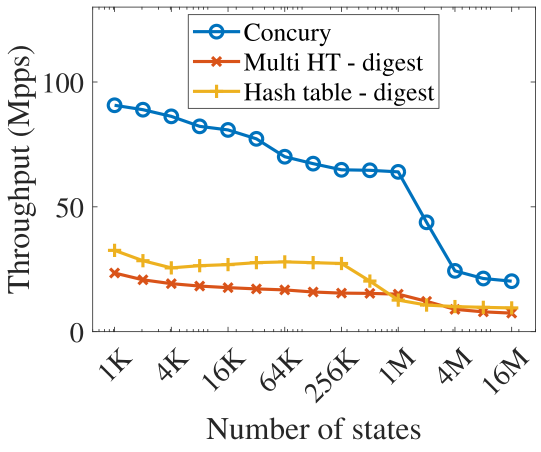
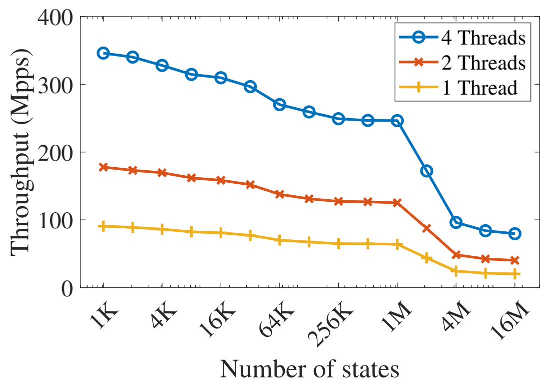


Figure 5.15: Throughput for DIP-V and Large network

5.6.2 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



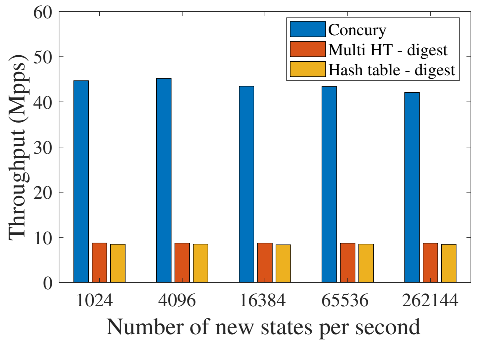


Figure 5.16: Throughput for multi-thread

Figure 5.17: Throughput during data plane updates

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. 5.2. 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

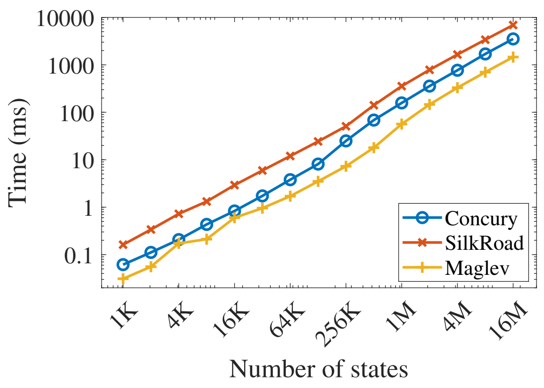


Figure 5.18: Time to insert new states to control plane

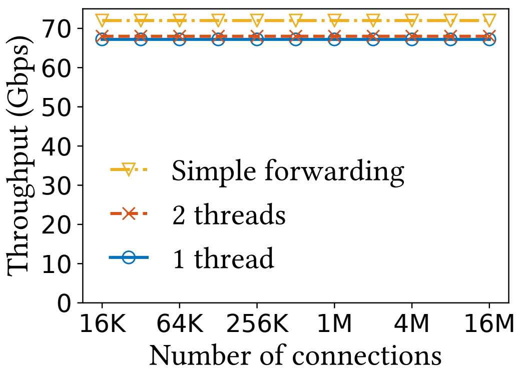


Figure 5.19: Throughput on DPDK

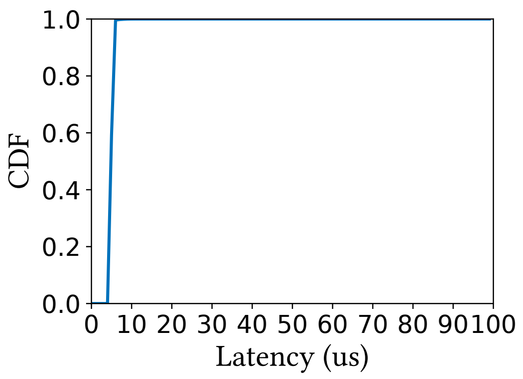


Figure 5.20: CDF of processing latency on DPDK

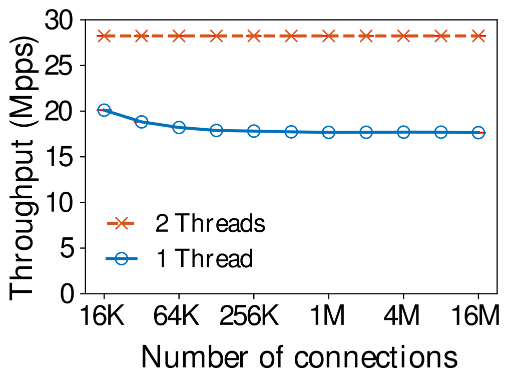


Figure 5.21: Throughput on DPDK in CloudLab

in a round-robin scenario, which provides the minimum cache hit ratio. We use 1883 lines of C++ code in total for this prototype.

Memory efficiency. Fig. 5.8 and 5.9 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 Maglevwhen 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. 5.10 and 5.11. 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. 5.12 and 5.13 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. 5.14 and 5.15, 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. 5.16 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 [37, 67]. 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. 5.17. 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. 5.15. 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 the change. Concury-DP provides a tremendous advantage in response time by leveraging OthelloMap, as shown in Fig. 5.6. 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. 5.18. 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.

5.6.3 Evaluation of Concury in real networks

Implementation details. We implement Concury as a software LB using Intel Data Plane Development Kit (DPDK) [5] 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 [5]. DPDK is useful for bypassing the complex networking stack in the Linux kernel, and it has utility functions for huge-page memory allocation and lockless FIFO, etc. We modify the code of Concury-DP and link it with DPDK libraries.

5.6.4 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 back-end servers (DIPs) in the cloud, and Node 2 works as the software LB. Node 1 uses the DPDK official packet generator Pktgen-DPDK [6] to generate random packets and sends them to Node 2. The 5-tuples 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 § 5.6.2.

Fig. 5.19 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 [37]. 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.[1] We evaluate up to 16M concurrent connections in the LB, as shown in Fig. 5.19. 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. 5.20 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.

[1]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 [37] deployed by Google shows that its maximum capacity on a 10GbE NIC is 12 Mpps (=6.14 Gbps).

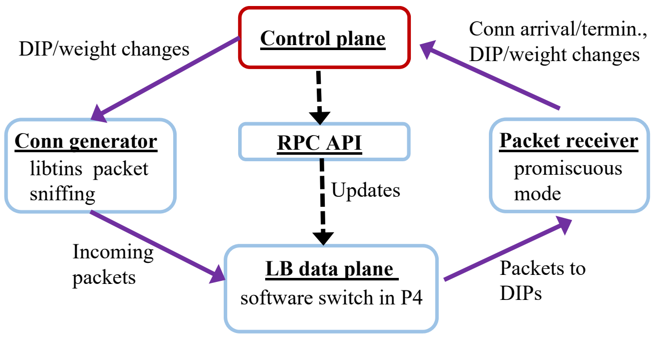


Figure 5.22: P4 prototype on Mininet

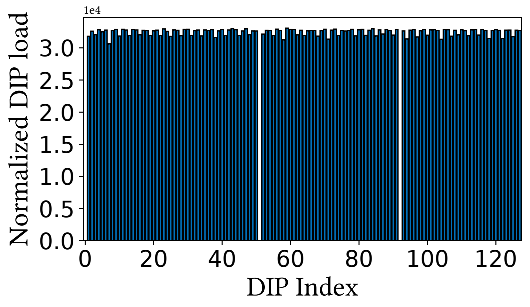


Figure 5.23: Normalized DIP load by P4 (real traffic)

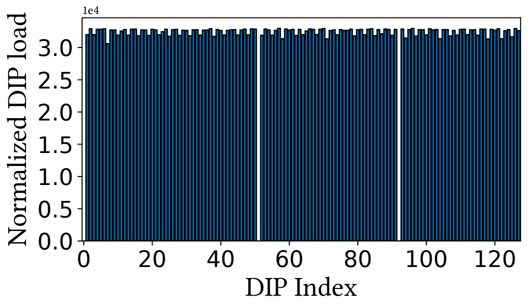


Figure 5.24: Normalized DIP load by P4 (synthetic traffic)

5.6.5 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 PCIe V3.0 connections between the CPU and the NIC. The switches between the two nodes support OpenFlow [69] and are claimed to provide full bandwidth. Fig. 5.21 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, the hash table based method cannot achieve the network capacity by 2 threads.

5.6.6 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 [20] of the $P4_{16}$ language [12]. 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 [21].

We use Mininet [17] 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. 5.22. 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 § 5.6.2 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. 5.23 and 5.24 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.

5.6.7 Summary of evaluation

As stated in § 5.3, 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. 5.18. 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.

Chapter 6

Future Directions

In this chapter, I list the ongoing works and possible research directions based on this dissertation.

Cloud and Edge Networks: Efficiency, Scalability, and Availability. My ongoing works go further in cloud and edge networking algorithm optimizations by the following directions: **1)** Further reuse the common computations and in-memory data structures of different algorithms collocating in the same forwarding middlebox (*e.g.,* load balancer, switch, or router) for higher forwarding efficiency as well as smaller memory footprint. **2)** Optimize the memory and computation costs and balance load where the data distributions or data retrieval patterns may be skewed, based on application-specific data structures and algorithms. **3)** Solve the data placement and retrieval problem other than hashing or virtual locations to be more flexible on the variety of data placement policies/requirements by the service provider or users. **4)** Develop an efficient and accurate virtual memory management and memory object allocation and indexing system for serverless or hardware-disaggregated environments.

I plan to solve another fundamental forwarding problem in data-centric applications where requires a co-design of the application and network: *forwarding based on configurable semantics rather than IP/MAC addresses.* For example, in the backend of web services inside a single cloud, the message distribution between end hosts relies on data exchange service middleware, and the prevailing programming model for data exchange is topic based publication-subscription (Pub-Sub), *e.g.,* RabbitMQ, Kafka, *etc.*. The existing Pub-Sub solutions introduce *data brokers* to collect messages from publishers, category messages via topics, and distribute messages to the subscribers of the corresponding topics. The data distribution and associated QoS requirements, however, can be achieved in a more efficient way by exploring the complete programmability of the SDN enabled switches on two key missing features in the current network layer and transport layer: 1) multicast based on topics and the subscription information instead of a set of IP addresses and 2) QoS policies are directly configured to the network based on topics and participant roles, instead of embedding into each packet.

Security and privacy enforcement and enhancement from resource limited devices. A great amount of data will be generated, processed, and transmitted by IoT devices (*e.g.,* wearable health care sensors, indoor hygrometers and thermometers, Alexa echo, *etc.*). Due to the constraints in computation power and memory space, many existing security solutions are not applicable to IoT devices. We develop VERID [**IoTDI'19**], a verifiable data outsourcing system for IoT applications. VERID enables important ranged selection and aggregate queries of sensing data while imposing minimal overhead for resource-constraint IoT devices. We then develop CCV [**INFOCOM'19**] to efficiently verify public-key certificates on resource-constrained IoT devices. Our ongoing work provides higher level security for certificate validation on IoT devices by supporting certificate revocation (CR). Existing on-device CR checking solutions either cost prohibitively much storage or require too much computation and network traffic when the CR list updates (submitted to **SIGMETRICS'21**).

Verify control plane configurations in virtualized networks. This is an ongoing work with Google, and more details are hidden here due to policies. Will submit to a USENIX conference. Modern cloud platforms feature a set of virtual network functions (VNFs) to facilitate the network deployment, including router, firewall, *etc.* Though even simpler in configuration and deployment compared with SDN, virtualized components still exhibit subtle behavior features which once encountered, may be hard to analyze and reinstate. Existing efforts present in network intent verification where a series of user-specified checks can be launched against the current network control plane or data plane to find out whether the current network contains a forwarding loop, drops traffics that should be forwarded, *etc.* This work fills the gap between the configuration and verification by providing a derivation engine to figure out the new data plane based on the current one and a proposed set of configuration changes. Based on this, more formal methods are used to verify the network behavior model to check if adding a new type of VNF, or changing the behavior of an existing VNF will cause ambiguity or break existing forwarding behaviors.

Platforming for Trusted Execution Environments (TEEs). TEEs from modern CPU manufacturers like Intel and ARM enable *secure remote execution.* Compared with secure multi-party computation (MPC), TEEs are much more efficient because no complex interaction is required during the computation, and TEEs support a wider range of applications because besides pure computation, TEEs can also store states (*e.g.,* store the votes and open the results later) and cause side-effects (POST to or GET from a website). Despite the advantages, TEEs are still not well-received by end users and the industry, mainly due to the following three problems I am solving (will submit to a USENIX conference): 1) the non-trivial hardware requirements and heavy configuration burdens for ordinary users; 2) no common code base and sharing platform like GitHub, PyPI, and DockerHub, and developers should work on their own for each feature; 3) some public libraries or applications are not publicly tested and thus no guarantee on the correctness and security requirements during execution. I plan to solve the above issues by introducing a whole new development platform, from environment check and setup tools, CLI package management, IDE semantic supports, code hosting, public verifiable unit-testing, and hosting of TEEs for resource limited devices and applications based on ad-hoc networking.

Chapter 7

Conclusion

This dissertation provides algorithmic and system innovations for cloud networks, with emphasis on efficiency, scalability, and flexibility. The contributions are categorized into the following three directions.

We provide 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.

Ludo hashing is a practical solution for space-efficient, fast, and dynamic key-value lookup engines that can fit into fast memory. Its core idea is to use perfect hashing and resolve the hash collisions by finding the seeds of collision-free hash functions instead of storing the keys. We present the detailed design of Ludo hashing, including the lookup, construction, and update algorithms under concurrent reading and writing. The analytical and experimental results show that Ludo hashing costs the least memory among known solutions that can be used for in-memory key-value lookups while satisfying $>$ 65 million queries per second for 1 billion key-value items on a single node. Ludo allows fast updates. We further demonstrate that Ludo hashing achieves high performance in practice by implementing it in two working systems deployed in CloudLab.

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.

Bibliography

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

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

[3] Implementation of Ludo Hashing in C++. `https://github.com/QianLabUCSC/Ludo`.

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

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

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

[7] 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`.

[8] Intel SSE4 Programming Reference. `https://goo.gl/J4HkVo`, 2007.

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

[10] Making facebook's software infrastructure more energy efficient with autoscale. https://goo.gl/692u64, 2014.

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

[12] $P4_{16}$ language. https://goo.gl/wp6no2, 2016.

[13] A10. https://www.a10networks.com/, 2017.

[14] Data Sharing on traffic pattern inside Facebook's datacenter network. https://research.fb.com/data-sharing-on-traffic-pattern-inside-facebooks-datacenter-network/, 2017.

[15] Libtins network packet sniffing and crafting library. https://goo.gl/36qokt, 2017.

[16] Loadbalancer.org inc. https://www.loadbalancer.org/, 2017.

[17] Mininet. http://www.mininet.org/, 2017.

[18] Netscaler, citrix systems inc. https://goo.gl/STMuUY, 2017.

[19] Nginx. https://www.nginx.com/, 2017.

[20] P4 Behavioral Model. https://goo.gl/vzBLE4, 2017.

[21] PI Library. `https://goo.gl/8Np9HQ`, 2017.

[22] Anonymous Source Code of the Concury Prototype. `https://www.dropbox.com/s/ruou2l340uu1f4u/concury%20code.zip`, 2019.

[23] H. Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic routing in future data centers. In *Proc. of ACM SIGCOMM*, 2010.

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

[25] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the Edge: Transport Affinity without Network State. In *Proc. of USENIX NSDI*, 2018.

[26] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. of ACM SODA*, 2009.

[27] Djamal Belazzougui and Fabiano C. Botelho. Hash, displace, and compress. In *Proc. of Algorithms-ESA*, 2009.

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

[29] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines. In *Proc. of ACM SIGCOMM*, 2014.

[30] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[31] Julie Anne Cain, Peter Sanders, and Nick Wormald. The Random Graph Threshold for k-orientiability and a Fast Algorithm for Optimal Multiple-Choice Allocation. In *Proc. of ACM-SIAM SODA*, 2007.

[32] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki Hitachi, John B. Vicente, and Daniel Villela. A survey of programmable networks. *SIGCOMM Computer Communication Review*, 1999.

[33] Denis Charles and Kumar Chellapilla. Bloomier Filters: A Second Look. In *Proc. of European Symposium on Algorithms*, 2008.

[34] Denis Charles and Kumar Chellapilla. Bloomier Filters: A Second Look. In *Proc. of ESA*, 2008.

[35] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA*, pages 30–39, 2004.

[36] David Chou et al. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proc. of ACM SOSP*, 2019.

[37] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*, 2016.

[38] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. A cool and practical alternative to traditional hash tables. In *Proc. 7th Workshop on Distributed Data and Structures (WDAS'06)*, 2006.

[39] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. Technical report, 2019.

[40] Bin Fan, Dave Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*, 2013.

[41] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael 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.

[42] Bin Fan, Dong Zhou, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen.

When cycles are cheap, some tables can be huge. In *Proc. of USENIX HotOS*, 2013.

[43] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 2000.

[44] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: An intellectual history of programmable networks. *ACM Queue*, 2013.

[45] Daniel Fernholz and Vijaya Ramachandran. The k-orientability Thresholds for $G_{n,p}$. In *Proc. of ACM/SIAM SODA*, 2007.

[46] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. In *Proc. of ACM/SIAM SODA*, 2011.

[47] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. 2014.

[48] Pu Gao and Nicholas C. Wormald. Load balancing and orientability thresholds for random hypergraphs. In *Proc. of ACM STOC*, 2010.

[49] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast Scalable Construction of (Minimal Perfect Hash) Functions. In *Proceedings of the International Symposium on Experimental Algorithms*, 2016.

[50] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, 2009.

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

[52] Chi-Yao Hong et al. Achieving High Utilization with Software-Driven WAN. In *Proceedings of ACM Sigcomm*, 2013.

[53] Sourabh Jain, Yingying Chen, Saurabh Jain, and Zhi-Li Zhang. VIRO: A Scalable, Robust and Name-space Independent Virtual Id ROuting for Future Networks. In *Proc. of IEEE INFOCOM*, 2011.

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

[55] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. 1997.

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

[57] Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo

hashing in hardware. In *Proceedings of the Forty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, volume 75, 2007.

[58] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 2009.

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

[60] James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *Proc. of IEEE S&P*, 2017.

[61] Marc Lelarge. A new approach to the orientation of random hypergraphs. . In *Proc. of ACM-SIAM SODA*, 2012.

[62] X. Li, D. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys*, 2014.

[63] Xiaozhou Li, Dave Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of ACM EuroSys*, 2014.

[64] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of ACM SOSP*, 2011.

[65] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 2015.

[66] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A Family of Perfect Hashing Methods. *The Computer Journal*, 1996.

[67] Rui Mao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[68] Martn Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 2007.

[69] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[70] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

[71] 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.

[72] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A Survey of Software-Defined Networking: Past, Present,

and Future of Programmable Networks . *IEEE Communications Surveys and Tutorials*, 2014.

[73] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *Proc. of USENIX NSDI*, 2018.

[74] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

[75] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2015.

[76] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[77] 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. Ananta: Cloud scale load balancing. 2013.

[78] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*, 2015.

[79] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. Emoma: Exact match in one memory access. *IEEE Transactions on Knowledge and Data Engineering*, 2017.

[80] Chen Qian and Simon Lam. ROME: Routing On Metropolitan-scale Ethernet . In *Proceedings of IEEE ICNP*, 2012.

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

[82] Martin Raab and Angelika Steger. Balls into Bins – A Simple and Tight Analysis. In *Lecture Notes in Computer Science*, 1998.

[83] Dipankar Raychaudhuri, Kiran Nagaraja, and Arun Venkataramani. MobilityFirst: A Robust and Trustworthy MobilityCentric Architecture for the Future Internet. *Mobile Computer Communication Review*, 2012.

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

[85] B. Schlinker et al. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proc. of ACM SIGCOMM*, 2017.

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

[87] S. Shi, C. Qian, and M. Wang. Re-designing Compact-structure based Forwarding for Programmable Networks. In *Proc. of IEEE ICNP*, 2019.

[88] Shouqian Shi, Chen Qian, Ye Yu, Xin Li, Ying Zhang, and Xiaozhou Li. Concury: A Fast and Light-weighted Software Load Balancer. *arXiv:1908.01889*, 2019.

[89] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5), 2016.

[90] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of ACM SIGCOMM*, 2002.

[91] Luis M. Vaquero and Luis Rodero-Merino. Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM CCR*, 2014.

[92] M. Wang, M. Zhou, S. Shi, and C. Qian. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. In *Proceedings of VLDB*, 2020.

[93] Minmei Wang et al. Collaborative Validation of Public-Key Certificates for IoT by Distributed Caching. In *Proc. of IEEE INFOCOM*, 2019.

[94] Minmei Wang, Chen Qian, Xin Li, and Shouqian Shi. Collaborative Validation of Public-Key Certificates for IoT by Distributed Caching. In *Proc. of IEEE INFOCOM*, 2019.

[95] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. Vacuum filters:

more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment*, 2019.

[96] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. 2011.

[97] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, 2006.

[98] Udi Wieder. *Hashing, Load Balancing and Multiple Choice*. Now Publishers, 2017.

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

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

[101] Tong Yang, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. Coloring Embedder: a Memory Efficient Data Structure for Answering Multi-set Query. In *Proc. of IEEE ICDE*, 2019.

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

[103] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog Computing: Platform and Applications. In *Proc. of IEEE HotWeb*, 2015.

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

[105] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. A concise forwarding information base for scalable and fast name lookups. In *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*, 2017.

[106] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. Othello Hashing for Scalable and Fast Name Switching. In *Proc. of IEEE ICNP*, 2017.

[107] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking*, 2018.

[108] Ye Yu, Xin Li, and Chen 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.

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

[110] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. Scaling up clustered network appliances with scalebricks. In *SIGCOMM*, 2015.

[111] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen.

Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of ACM CoNEXT*, 2013.

Appendix A

Appendix

A.1 Appendix for Re-designing Compact-structure based Forwarding for Programmable Networks

A.1.1 Not use the link-index design for BFW

One possible question is that whether the DP could maintain $\log_2 d$ BFs rather than d BFs. To test a key, each BF returns either 0 or 1. Hence, the replies from all $\log_2 d$ BFs can form a $\log_2 d$-bit long index representing a set of keys that should be forwarded to each link. We call this the link-index design. After careful study, we find that this method is very inefficient in memory cost and may cause high false positives. The key insight is that the length of a BF grows approximately linearly with the number of elements in the represented set, and the number of keys to be *False* does not contribute to the memory footprint. More specifically, although the number of Bloom filters in the current design is larger, the overall memory consumption is

$$\sum_{b=1}^{d} -n_i \cdot \frac{n_h}{ln(1 - FP_b^{-1/n_h})} = n \cdot \frac{n_h}{-ln(1 - FP_b^{-1/n_h})}$$

where n_i is the number of keys in the i-th Bloom filter. While for the link-index design, although the number of Bloom filters is smaller, its overall memory consumption is

$$\sum_{b=1}^{n_b} \frac{-n}{2} \cdot \frac{n_h}{ln(1 - FP_b^{-1/n_h})} = \frac{n \cdot n_b}{2} \cdot \frac{n_h}{-ln(1 - FP_b^{-1/n_h})}$$

As shown above, the memory consumption of the link-index design is linearly proportional to n_b and is $n_b/2$ times larger than that of the current design. This problem is significant, especially in L7 networks, where the number of links in every node could be very large. Based on the above comparison, we use the current design in BFW.

A.1.2 OFW emptiness indicator

The expected rate 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 $\epsilon_b = (\frac{m_b-1}{m_b})^{n_k} \approx e^{-\frac{n_k}{m_b}} \approx 0.368$. So the expected possibility to detect an alien key via emptiness test is approximately $1-(1-0.471)(1-0.368) \approx 0.666 \geq 0.5$. Based on that observation, we reserve the last bit of the fingerprint field as slot emptiness indicator. The lookup is considered as successful only when then both statements hold: 1) two slots are both non-empty, and 2) the key fingerprint matches the value in the calculated fingerprint.

A.1.3 Derivation of η

We found the exact analytical expression of η is very hard to derive because $\eta = \frac{n_c}{n_k}$, where n_c is the number of the keys that meet the requirement that for each one of them, there is no other key that shares the same fingerprint with it while shares one or two mapped buckets with it. The derivation of the analytical expression of n_c is intuitively as hard as exhaustively enumerating all possible placements of all the keys and calculating the weighted average over all these possible placements.

We evaluate this value via experiments. We identify the most important parameter to influence η is the fingerprint length l_d and the configuration of Cuckoo filter, n_s and n_b. We let the $n_s = 4$ and $n_b = 2$ to be the normal setting and change the digest length from 1 to 16 to observe the corresponding changes of η. The experiments are repeated 10 times, with different randomly sampled key sets. We also repeated the above experiment to let n_k range from $10K$ to $10M$, and the mean value, the higher bound, and the lower bound at each digest length are shown in Fig. A.1. The key size has neglect influence on η, as expected.

A.1.4 CFW cache line alignment

For the *aligned* setting, we define the function $f(i, l_s)$ as the expected memory bus read operations in a value lookup after i slots have been tried to match, where the slot length is l_s. l_s may be $l_d + l_p$ or $l_k + l_p$ for different levels.

$$f(i, l_s) = \lfloor i/n_s \rfloor \cdot \lceil n_s \cdot l_s / l_c \rceil + \lceil (i \mod n_s) \cdot l_s / l_c \rceil$$

The expected memory bus accesses for the *aligned* setting consists of three parts: 1) loading the current input key from the memory, 2) looking up the key in Level 1 (we are assuming here the key is located in all 8 possible slots with the same possibility), and 3) looking up the key in Level 2 similarly if Level 1 is a miss. Note here the valid key collisions are perfectly avoided by our design, so there is no other possible case.

$$\begin{aligned} E(C_m^c) &= \lceil l_k/l_c \rceil + E_{\eta,l_d} \sum_{i=1}^{n_b \cdot n_s} \frac{f(i, l_d + l_p)}{n_b \cdot n_s} + (1 - E_{\eta,l_d}) \\ &\quad \cdot \left(f(n_b \cdot n_s, l_d + l_p) + \sum_{i=1}^{n_b \cdot n_s} \frac{f(i, l_k + l_p)}{n_b \cdot n_s} \right) \qquad \text{(A.1)} \\ E(C_{m,e}^c) &= \lceil l_k/l_c \rceil + (f(n_b \cdot n_s, l_d + l_p) + f(n_b \cdot n_s, l_k + l_p)) \end{aligned}$$

For the *compact* setting, we assume the buckets is aligned to the cache line at the first element, which is easily achieved. Then the Greatest Common Divider (gcd) of the length of the cache line and the length of the bucket (both in bits) is the atomic step between all possible relative positions. We define l_s as the length of a slot and $l_b = n_s l_s$ as the length of a bucket, both in Level 1 or the 2nd level. Then $l_s = l_d + l_p$ at Level 1 and $l_s = l_k + l_p$ at the 2nd level. Then we define $l_g = gcd(l_c, l_b)$. We always need to access the memory at least $\lfloor l_b/l_c \rfloor$ times both in the *compact* setting and in the *aligned* setting. An extra memory read happens when the remainder of the must read bits ($l_b \mod l_c$ bits long) is distributed in two cache lines. Two typical examples are shown in Fig. A.2. We define a function $l_r(l)$ to denote the remainder of l bits:

$$l_r(l) = \lceil (l \mod l_c)/l_g \rceil \cdot l_g$$

Then we can derive the expected memory loads if we only access the first i slots in a bucket:

$$E(C_{m,i}) = \lceil (i \cdot l_s)/l_c \rceil + \begin{cases} \frac{l_r(i \cdot l_s) - l_g}{l_c}, & l_r(i \cdot l_s) \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

To determine M^c under the *aligned* setting, two factors e_{b1} and e_{b2} are defined as the expansion factors due to aligning buckets to cache lines for Level 1 and the 2nd level, respectively. And for the *compact* setting, we just let $e_{b1} = e_{b2} = 1$.

$$e^c_{b1} = \frac{\lceil n_s \cdot (l_d + l_p)/l_c \rceil \cdot l_c}{n_s \cdot (l_d + l_p)}$$

$$e^c_{b2} = \frac{\lceil n_s \cdot (l_k + l_p)/l_c \rceil \cdot l_c}{n_s \cdot (l_k + l_p)}$$

The memory footprint of both the *aligned* and *compact* DP FIB is:

$$\begin{aligned} M^c_f &= M_{1st\ level} = e_{b1} \cdot E_{\eta,l_d} \cdot n_k \cdot e_l(l_d + l_p) \\ M^c &= M_{1st\ level} + M_{2nd\ level} \\ &= M^c_f + e_{b2}(1 - E_{\eta,l_d})n_k \cdot e_l(l_k + l_p) \end{aligned} \tag{A.2}$$

A.1.5 Detailed time complexity analysis

BFW. C_m and C_h are equal for Bloom filters. As the starting index of the Bloom filters is uniformly randomly picked, the total queried Bloom filters will be $d/2$. For a hit in a single Bloom filter, evidently, we have the number of hash function invocations to be n_h because BFW checks all slots of this key to be 1. But for an alien lookup, it may end at the first empty slot. Precisely calculating the expected number of memory accesses is hard because we have to calculate the weighted average over all possible key - port distributions. Instead, we propose to derive the expected possibility p of every single slot in the Bloom filter array being *False* and assume p as ground truth to further derive other equations. Given n_h and m, p is easily expressed for a single Bloom filter.

$$p = (1 - \frac{1}{m})^{n_k n_h} \approx e^{-\frac{n_k n_h}{m}} \tag{A.3}$$

Under this approximation, the expected number of tests of an alien lookup for a single Bloom filter is represented by:

$$C_t = \left(\sum_{i=1}^{n_h - 1} i \cdot (1-p) \cdot p^{i-1} \right) + n_h p^{n_h - 1} \tag{A.4}$$

One should note that the two above equations are for a single Bloom filter with n_k keys and m slots. BFW has the "assembly" of Bloom filters. The key point here is that m always appears with n_k in the equations, which means we can ignore the difference between BFW and Bloom filter if we treat $\frac{m}{n}$ as a whole. In addition, when we want to meet a target false positive rate by freely adjusting m and n_h for a given n_k, the optimal values of m and n_h let p to be $\frac{1}{2}$. But it does not mean there is an optimal m for false positive rate. We can always get a lower false positive rate by making m larger.

Combine all equations above, the approximate expected number of memory accesses and hash function invocations per lookup are as follows, based on the assumption that the key collision rate is designed to be low such that for a valid key, we ignore the possibility to collide in the middle.

$$\begin{aligned}
E(C_h^b) &= \sum_{i=1}^{d-1} \frac{1}{d}\left((i-1)C_t + n_h\right) = \frac{d-1}{2}C_t + n_h \\
E(C_{h,e}^b) &= \left(\sum_{i=1}^{d-1} \left((1-p)^{n_h}\right)^{i-1} \cdot \left(1-(1-p)^{n_h}\right)(i \cdot C_t)\right) \\
&\quad + \left((1-p)^{n_h}\right)^d (d \cdot C_t)
\end{aligned} \tag{A.5}$$

$$\begin{aligned}
E(C_m^b) &= E(\lceil l_k/l_c \rceil + C_h^b) = \lceil l_k/l_c \rceil + E(C_h^b) \\
E(C_{m,e}^b) &= E(\lceil l_k/l_c \rceil + C_{h,e}^b) = \lceil l_k/l_c \rceil + E(C_{h,e}^b)
\end{aligned} \tag{A.6}$$

CFW. The situation is more complex for a CFW to evaluate C_m and C_h. The two numbers are intuitively equal, but many exceptions also exist. Furthermore, the key locations may vary for different implementations. We just assume the simplest and most general case: the keys are fixed in length and are directly embedded into the slots. Assuming the key locations are uniformly random, the expected numbers of hash function invocation for valid keys and alien keys are easily derived as follows:

$$\begin{aligned}
E(C_h^c) &= E_{\eta,l_d}\left(1 + \sum_{i=1}^{n_b \cdot n_s} \frac{\lceil (i-1)/n_s \rceil + 1}{n_b \cdot n_s}\right) \\
&\quad + (1 - E_{\eta,l_d})\left(1 + n_b + \sum_{i=1}^{n_b \cdot n_s} \frac{\lceil (i-1)/n_s \rceil + 1}{n_b \cdot n_s}\right) \\
&= 2 + \frac{n_b - 1}{2} + (1 - E_{\eta,l_d})(1 + n_b) \\
E(C_{h,e}^c) &= 1 + n_b + n_b = 1 + 2n_b
\end{aligned} \tag{A.7}$$

To derive $E(C_m)$ for CFW DP, we identify an important value $E(C_{m,i})$ – the expected number of memory loads for the first i slots in any bucket. The detailed derivation of it is in Appendix A.1.4. Therefore, the expected memory reads for a whole bucket $E(C_b) = E(C_{m,n_s})$. So the expected number of memory bus accesses is:

$$\begin{aligned} E(C^c_m) &= \lceil l_k/l_c \rceil + \sum_{i=1}^{n_b \cdot n_s} \frac{\lfloor i/n_s \rfloor \cdot E(C_b) + E(C_{m,i})}{n_b \cdot n_s} \\ E(C^c_{m,e}) &= \lceil l_k/l_c \rceil + n_b \cdot E(C_b) \end{aligned} \tag{A.8}$$

OFW. The number of hash function invocations for a valid key is always 3. An alien key is detected when 1) the slots are marked empty 2) the fingerprints does not agree.

$$\begin{aligned} C^o_h &= 3 \\ C^o_{h,e} &= \epsilon_a + 2 \cdot (1-\epsilon_a)\epsilon_b + 3 \cdot (1-\epsilon_a)(1-\epsilon_b) \end{aligned} \tag{A.9}$$

C_m equals C_h for most of the cases in an OFW, except for cases where the slots being read runs across a cache line border. As the slots are very short in length, the possibility of slots existing in two consecutive cache lines are rather low. To quantify the extra memory load possibility, we define $l_g = gcd(l_d + l_p, l_c)$. Similar with the derivation in Appendix A.1.4, assuming the $l_d + l_p$ is always smaller than l_c, the C_m and $C_{m,e}$ are expressed as follows:

$$\begin{aligned} E(C^o_m) &= \lceil l_k/l_c \rceil + 2 \cdot \left(1 + \frac{l_d + l_p - l_g}{l_c}\right) \\ E(C^o_{m,e}) &= \lceil l_k/l_c \rceil + ((1-\epsilon_a)\epsilon_b + 2 \cdot (1-\epsilon_a)(1-\epsilon_b)) \\ &\quad \cdot \left(1 + \frac{l_d + l_p - l_g}{l_c}\right) \end{aligned} \tag{A.10}$$

A.1.6 Detailed analysis for collision and false positive rates

BFW. We continue to use the previous approximation for simplicity: the possibility of a Bloom filter slot being *False* is p as shown in Equation (A.3). Then the

false positive rate of alien keys FP is calculated as:

$$FP^b = 1 - (1 - (1 - p)^{n_h})^d \tag{A.11}$$

Although key collisions are different from false positives, a valid key at a Bloom filter not containing it is the same as an alien key at this Bloom filter. So the **expected valid key collision rate per lookup** is calculated as:

$$\begin{aligned} E(CR^b) &= \sum_{i=1}^{d} \left(1 - \left(\frac{1}{2}\right)^{n_h}\right)^{i-1} \cdot \left(\frac{1}{2}\right)^{n_h} \cdot \frac{1}{d} \\ &= \sum_{i=1}^{d} \left(1 - \frac{1}{2^{n_h}}\right)^{i-1} \cdot \frac{1}{d \cdot 2^{n_h}} \\ &\approx \sum_{i=1}^{d} \left(1 - \frac{i-1}{2^{n_h}}\right) \cdot \frac{1}{d \cdot 2^{n_h}} = \frac{1}{2^{n_h}} - \frac{d-1}{2^{2n_h+1}} \end{aligned} \tag{A.12}$$

Note that the approximation in Equation (A.12) is valid only when $2^{n_h} \gg d$, which may be true in some L2 networks where the desired false positive is low and the number of ports at a forwarder is small. The approximations work as references, we will not use them in the following discussions.

CFW. As we are intentionally avoiding valid key collisions, the **valid key collision rate per lookup** is always 0.

$$CR^c = 0 \tag{A.13}$$

As Level 2 stores full keys, false positives only happen at Level 1 because of possible fingerprint collisions. The **false positive rate per alien key lookup** FP can be analytically expressed under the assumption that each key is located in all its 8 possible slots with the same possibility.

$$FP^c = 1 - (1 - \frac{1}{2^{l_d}})^{r_l E_{\eta,l_d} \cdot n_s n_b} \tag{A.14}$$

OFW. As the cases for the alien key to have a return value are where both the retrieved slots are not marked as empty, and the calculated fingerprint field matches with the fingerprint of the alien key. The **alien key false positive rate per lookup** FP^o is expressed as:

$$FP^o = \frac{1}{2^{l_d-1}} \cdot (1-\epsilon_a) \cdot (1-\epsilon_b) \tag{A.15}$$

There is not valid key collisions according to the loop-free structure of Othello.

$$CR^o = 0 \tag{A.16}$$

A.1.7 Achieving minimal memory to meet target false positive rate

For BFW, we let m grow one bit by one bit until hitting the target false positive. As the memory and false positive rate are all determined by l_d in an OFW, we grow the l_d of OFW to record the first l_d meeting the current target false positive. The l_d in CFW, however, is not monotonically related to memory footprint and the false positive rate. This is because l_d will influence the portion of keys located in Level 1 η, and thus a smaller l_d leads to a memory growth in the 2nd level, which may be more than the memory footprint increment of increasing l_d by 1. As the numbers of ports d on a forwarder and the length of keys l_k differ a lot in L2 networks and L7 overlay networks, we pick $l_p = 5$, $l_k = 128$ to simulate the L2 networks, and $l_p = 14$, $l_k = 360$ for the L7 networks.

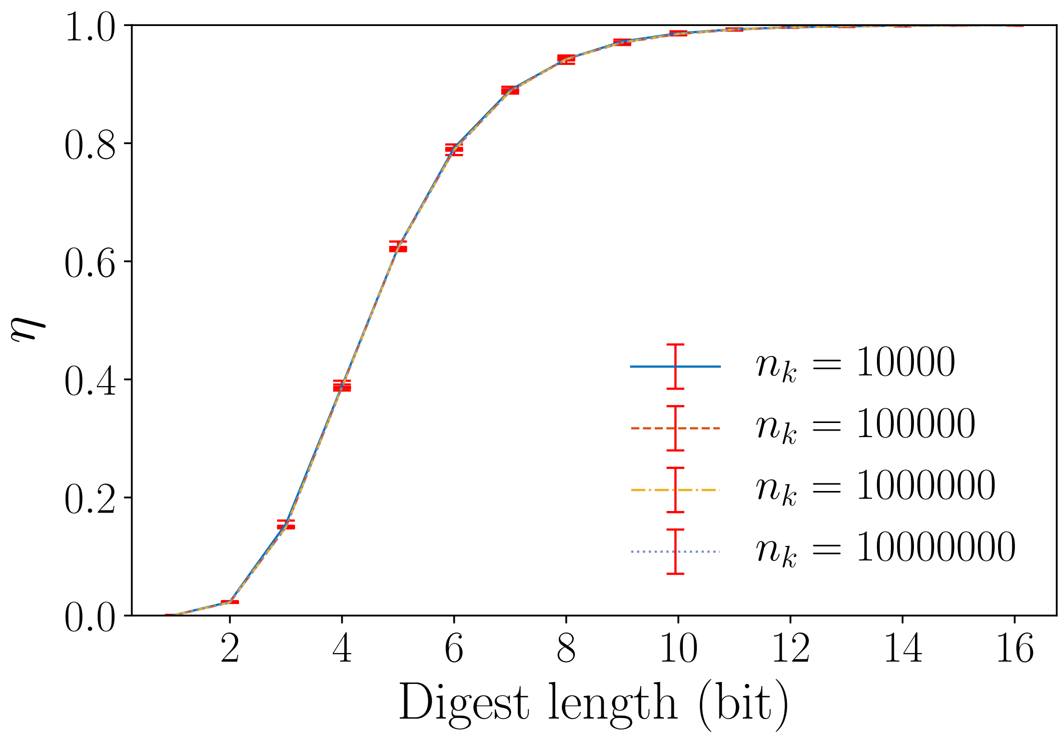


Figure A.1: The portion of keys at Level 1 CFW FIB

A.2 Appendix for Ludo hashing

A.2.1 Pseudocode

We also show the pseudocode of the `insertion` algorithm on Ludo maintenance program algorithm in Algorithm 2, the `insertion` on the Ludo lookup program in Algorithm 3, the concurrent `lookup` algorithm of Ludo in Algorithm 4, and the `construct` algorithm for Ludo lookup structure from the Ludo maintenance structure in Algorithm 5. Algorithm 6 shows the subroutine in Ludo control plane to find a seed for a bucket.

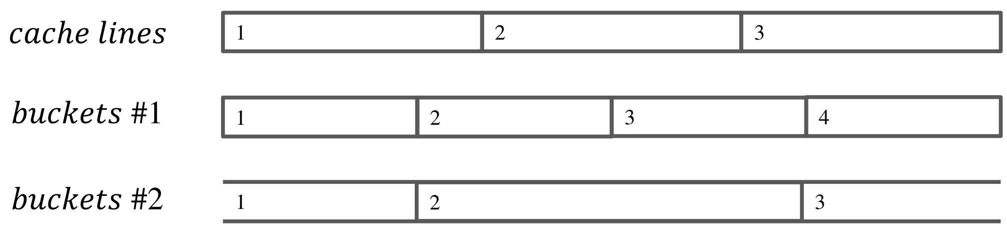


Figure A.2: Extra memory loads for *compact* setting

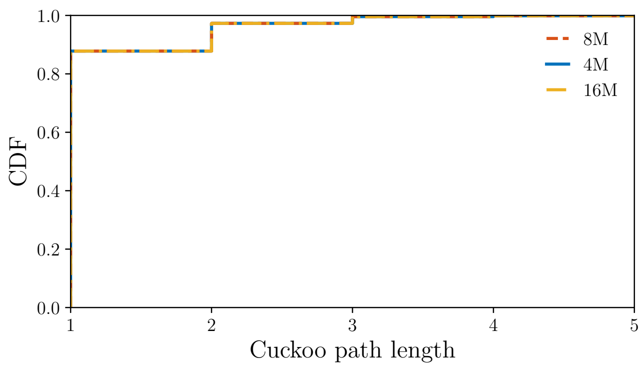


Figure A.3: CDF of Cuckoo path length

A.2.2 Load factor for successful insertions.

From existing theoretical results of random graphs, it has been proved by both [45] and [31] that if the average degree d of a random directed graph G of n vertices is no higher than a threshold d_k, then

$$\lim_{n \to \infty} \Pr(G \text{ is } k\text{-orientable}) = 1 \text{ if } d < d_k$$

We say G is asymptotically almost surely (a.a.s.) k-orientable. A graph is k-orientable if every vertex has in-degree at most k. Consider that each bucket of (2,4)-Cuckoo corresponds to a vertex of a random graph, and each key corresponds to an edge. A key stored in a bucket can be considered as an edge contributing to an in-degree to the vertex. Hence, a 4-orientable graph is equivalent to a (2,4)-Cuckoo where each bucket stores at most 4 keys. The above proved result [45, 31] is equivalent to the following statement. If the load factor is no higher than $d_4/8$ and the table is sufficiently large, all inserted keys can be stored in a (2,4)-Cuckoo such that every bucket has at most 4 keys. The numerical value of d_4 is 7.843, meaning the threshold of the load factor can be as much as 0.9803, provided by both [45] and [31]. Many later studies confirm this result [48, 46, 61, 98]. Note the extreme cases in practice that cause failed insertions do not conflict with this theoretical result. In practice, the length of a cuckoo path is within a small constant. We show the experimental results of the lengths of cuckoo paths in Fig. A.3, for Ludo hashing with load factor $< 95\%$, 4, 8, and 16 million items, and 10 runs for each setup. We find that all lengths of the Cuckoo paths are $\leqslant 5$, and more than 95% are smaller than 3. In our design, we set the load factor threshold to be 95% due to practical issues such as the maximum number of steps of evictions in implementation. We have not observed a single failure among over 20 billion insertions during our tests.

A.3 Appendix for Concury

A.3.1 Example of consistency violation by static hashing.

Consider the example shown as Fig. A.4. Suppose the LB uses static hashing to evenly distribute traffic to four DIPs. Connection C_1, whose hash value is 0.3, is mapped to DIP_2. All packets of C_1 should be forwarded to DIP_2 if there is no DIP pool change. However, if there is a change of the DIP pool, e.g., the failure of DIP_4, then the hashing-to-DIP mapping needs to be adjusted for balancing. As a result, later packets of C_1 will be forwarded to DIP_1, causing a PCC violation. Other stateless hashing algorithms, such as consistent hashing, experience similar problems. These problems are more significant in edge networks where the state may be multi-connection and long-term.

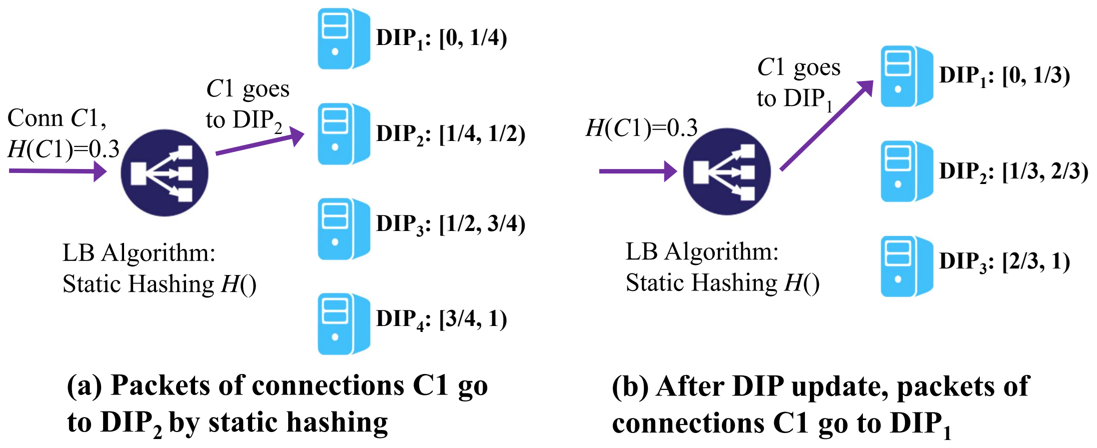


Figure A.4: PCC violation of static hashing

A.3.2 Pseudocode

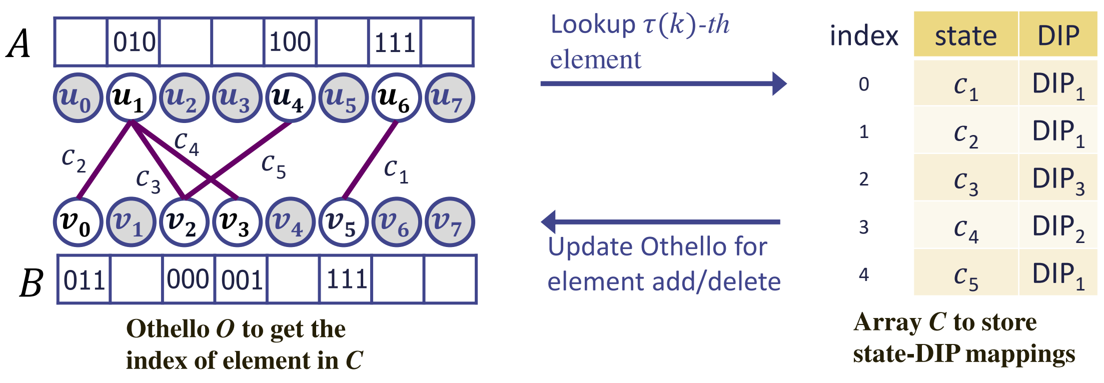
We also show the pseudocode of the Concury-DP lookup algorithm in Algorithm 7 and the Concury-DP updating algorithm in Algorithm 8.

A.3.3 Data plane complexity analysis and comparison.

Time cost. *1) Concury.* 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. The 6 read operations include 1 for the VIP array access, 1 for basic information of the Othello, 2 for Othello access, and 2 for finding the actual DIP of the *Dcode*. The time cost is the same for stateful and stateless packets. *2)Cuckoo+digest.* We compare Concury with the hash table plus digest approach, which is applied by some mainstream systems [37, 67]. We assume a (2,4) Cuckoo hash table, which has been shown as an optimized and up-to-date LB design choice of a hash table [67]. For a stateful packet, *on average* Cuckoo+digest needs 3.5 hash computations, including 2 for generating the 64-bit digest and 1.5 for locating the buckets (50% found in the first bucket and 50% found in the second bucket). It also takes 7 memory read operations on average: 1 for basic information of the hash table and 6 for hash table lookups (4 lookups per bucket). It takes 6 digest comparisons on average (4 per bucket). For a stateless packet, Cuckoo+digest needs to read and compare the key digest to all slots in the two buckets. Hence it takes 4 hash computations, 9 memory read operations, and 8 digest comparisons. Concury is faster compared to Cuckoo hashing based solutions, as shown in Table A.1.

Space cost. *1) Concury.* Let n be the number of total states, and l_d is the length of *Dcode*, assuming all Othellos use the same length of *Dcode*. The Othellos take $2.33l_d \cdot n$ bits, the VIP array takes $64m$ bits, and the DIP array takes $2^{l_d}l_vm$ bits where l_v is the length of the DIP index. A DIP and port take 48 bits. The total space cost is $2.33l_dn + 64m + 2^{l_d}l_vm + 48 \cdot 2^{l_v}$ bits. *2) Cuckoo+digest.* Assume the hash table load factor is 90%, Cuckoo+digest takes $1.1(64 + l_v) \cdot 4n$ for the hash table, $2^{l_d}l_vm$ for the weighted load balancer [37], and $48 \cdot 2^{l_v}$ for the DIP retrieval table. Hence the total



A
010
100
111
u_0 u_1 u_2 u_3 u_4 u_5 u_6 u_7
c_2
c_4
c_3
c_5
c_1
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7
B
011
000
001
111
Othello O to get the index of element in C
Lookup $\tau(k)$-th element
Update Othello for element add/delete
Array C to store state-DIP mappings

index	state	DIP
0	c_1	DIP_1
1	c_2	DIP_1
2	c_3	DIP_3
3	c_4	DIP_2
4	c_5	DIP_1

Figure A.5: OthelloMap of 5 state-DIP mappings

is $1.1 \cdot (64 + l_v)n + 2^{l_d} l_v m + 48 \cdot 2^{l_v}$ bits. Since $n \gg m$, to compare the space cost of Concury and Cuckoo+digest is mainly comparing $2.33 l_d n$ and $1.1(64+l_v)n$. In practical settings, $2.33 l_d n$ is much smaller than $1.1 \cdot (64 + l_v)n$. For example, using $l_d = 12$ and $l_v = 12$, $2.33 l_d n = 28n$ and $1.1 \cdot (64 + l_v)n = 83.6n$. The experimental results show that the Cuckoo+digest method (Maglev) needs around 3x memory compared to Concury, which agrees with the analysis here. Note that one assumption here is that for all VIPs, the length of *Dcode* is the same.

A.3.4 Example of OthelloMap.

Fig. A.5 shows an example of OthelloMap for one VIP. Array C stores all current state-DIP mappings. OthelloMap also maintains an Othello structure to reflect the index of each mapping in the array. For example, state c_1 is stored at index 0 of the array. Hence the lookup result of c_1 in the Othello O is $111 \bigoplus 111 = 0$. Once a new state-DIP mapping is inserted or an expired mapping is deleted from C, O should change accordingly. If an existing mapping, say c_2 to DIP_1 at index 1 is deleted, the mapping as the last element, i.e., c_5 to DIP_1 should be moved to index 1.

Input: The Ludo maintenance structure $\langle O_M, C\rangle$ and the item to insert $\langle k, v\rangle$

Result: The insertion message $\langle val, update\_seq, failed\_key\rangle$ for Ludo lookup program

```
begin
 1    $val \leftarrow v$
 2    $update\_seq \leftarrow$ new empty list
      // I: Insert item and record cuckoo path
 3    $cuckoo\_path \leftarrow C.Insert(k, v)$
 4    if $cuckoo\_path$ is empty then
 5        Insert to fallback table
 6        $failed\_key \leftarrow k$
 7        return
 8    for $position$ in $cuckoo\_path$ do
 9        $bIdx, sIdx \leftarrow position$
10        $b \leftarrow C.buckets[bIdx]$
11        $k \leftarrow b.keys[sIdx]$
12        // II: Reverse the bucket locator record, and record the
             influenced bits in Othello
13        $Ochg \leftarrow O.Insert(k, 1 - O.LookUp(k))$
          // III: Find a new seed
14        $b.s \leftarrow$ FindSeed $(b)$
15        $vorder \leftarrow$ Order of the values based on $b.s$
16        $update\_seq.add(\langle bIdx, sIdx, b.s, vorder, Ochg\rangle)$;
      end
end
```

Algorithm 2: `insertion` algorithm on Ludo maintenance program

```
Input: Ludo lookup structure $\langle O_L, T\rangle$ and the insertion message
       $\langle val, update\_seq, failed\_key\rangle$, the version array $V$, a global lock $L$
       for fallback
Result: Ludo lookup structure is updated
begin
    if $failed\_key$ is set then
        $L.lock()$
        Insert to fallback table
        $L.unlock()$
        return
    for $i = update\_seq\_size - 1, \cdots, 3, 2, 1, 0$ do
        $bIdx, sIdx, s, vorder, Ochg \leftarrow update\_seq[i]$
        // I: Copy current bucket
        $b \leftarrow$ copy of $T.buckets[bIdx]$
        // II: Update the temporary bucket
        $b.s = s$
        Order values in $b$ according to $vorder$
        // III: Consistency under concurrent R/W
        $V[bIdx \mod 8192] \leftarrow V[bIdx \mod 8192] + 1$
        compiler barrier
        Othello atomic update ($Ochg$)
        $C.buckets[bIdx] \leftarrow b$
        compiler barrier
        $V[bIdx \mod 8192] \leftarrow V[bIdx \mod 8192] + 1$
    end
end
```

Algorithm 3: `insertion` on the Ludo lookup program

```
Input: Ludo lookup structure, the version array $V$, and the key $k$ to
       look up
Output: The query result $v$
begin
    // Never entered in practice, under 95% load.
    if Fallback table has entries then
        $L.lock()$
        $v \leftarrow$ read from fallback table
        $L.unlock()$
        return
    while true do
        // Enssure bucket versions are even
        $v_0, v_1 \leftarrow V[h_0(k) \mod 8192], V[h_1(k) \mod 8192]$
        compiler barrier
        if $v_0$ or $v_1$ is odd then continue
        // Atomically query bucket locator
        $l \leftarrow$ Othello atomic lookup $(k)$
        // Fetch the bucket holding $k$
        $b \leftarrow h_l(k)$-th bucket of the table
        // Enssure versions have not changed
        compiler barrier
        $v'_0, v'_1 \leftarrow V[h_0(k) \mod 8192], V[h_1(k) \mod 8192]$
        if $v_0 \neq v'_0$ or $v_1 \neq v'_1$ then continue
        // Fetch the value of $k$
        $s \leftarrow$ slot locator seed stored in $b$
        $v \leftarrow b.slots[\mathcal{H}_s(k) \mod 4]$
        break
    end
end
```
Algorithm 4: Concurrent `lookup` algorithm on the Ludo lookup structure

```
Input: The Ludo maintenance structure $\langle O_M, C \rangle$
Output: The Ludo lookup structure $\langle O_L, T \rangle$
begin
    // I: Othello maintenance to lookup
    $O_L \leftarrow O_M$ converts to a lookup structure
    // II: New empty (2,4)-Cuckoo Hash Table
    $T \leftarrow$ empty table of size $C.size$
    for $i = 1, 2, 3, \cdots, C.bucket\_size$ do
        $b \leftarrow C.buckets[i]$
        $b' \leftarrow T.buckets[i]$
        // III: Copy locator seeds
        $s \leftarrow b.seed$
        $b'.seed \leftarrow s$
        // IV: Copy values to target buckets
        for $\langle k, v \rangle$ in valid items of $b$ do
            $sidx \leftarrow \mathcal{H}_s(k) \mod 4$
            $b'.values[sidx] \leftarrow v$
        end
    end
end
```

Algorithm 5: `construct` algorithm for Ludo lookup structure from the Ludo maintenance structure

```
Input: The Ludo maintenance structure bucket $b$
Input: The new seed $s$
begin
    for $s = 0, 1, 2, \cdots$ do
        $taken \leftarrow$ 4-element boolean array
        $success \leftarrow$ true
        for $k$ in valid keys of $b$ do
            $sid \leftarrow \mathcal{H}_s(k)$
            if $taken[sid]$ then
                $success \leftarrow$ false
                break
            $taken[sid] \leftarrow$ true
        end
        if $success$ then
            return s
        end
    end
end
```

Algorithm 6: Subroutine `FindSeed`

```
Input  : VIP index $i$, 5-tuple $t$, hash func. $h_a$ and $h_b$
Output: DIP $d$
1 $A_i \leftarrow VIPArray[i]$;
  // $A_i$: memory address of array $A$ of the $i$-th Othello
2 $B_i \leftarrow A_i + m_a$;
  // $B_i$: memory address of array $B$ of the $i$-th Othello
3 $Dcode \leftarrow A_i[h_a(t)] \oplus B_i[h_b(t)]$;
4 $d \leftarrow DA[i][Dcode]$;
```
Algorithm 7: Data plane lookup algorithm of Concury

```
Input  : $\langle i, A', B', DA' \rangle$ from update message
// $i$: VIP index; $A'$: new array $A$; $B'$: new array $B$, $DA'$:
   new $DA$ in dimension $i$
1 $A_i \leftarrow VIPArray[i]$;
2 $B_i \leftarrow A_i + m_a$;
3 $ArrayCopy(A_i, A')$; $ArrayCopy(B_i, B')$;
4 $ArrayCopy(DA[i][], DA')$;
  // $ArrayCopy$ copies received arrays to existing ones with
     concurrent control.
```
Algorithm 8: Data plane update algorithm of Concury

LB Algorithm	#Hashes	#Reads	Other computation
Cuckoo+digest [67, 37] stateless pkts	4	9	cmpr digest 8 times
Cuckoo+digest [67, 37] stateful pkts	3.5	7	cmpr digest 6 times
Concury	2	6	1 XOR

Table A.1: Data plane time cost breakdown (per lookup)