Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing

Ye Yu, *Student Member, IEEE and ACM,* Djamal Belazzougui, Chen Qian, *Member, IEEE and ACM,* and Qin Zhang

***Abstract*—Network algorithms always prefer low memory cost and fast packet processing speed. Forwarding information base (FIB), as a typical network processing component, requires a scalable and memory-efficient algorithm to support fast lookups. In this paper, we present a new network algorithm, Othello Hashing, and its application of a FIB design called Concise, which uses very little memory to support ultra-fast lookups of network names. Othello Hashing and Concise make use of minimal perfect hashing and relies on the programmable network framework to support dynamic updates. Our conceptual contribution of Concise is to optimize the memory efficiency and query speed in the data plane and move the relatively complex construction and update components to the resource-rich control plane. We implemented Concise on three platforms. Experimental results show that Concise uses significantly smaller memory to achieve much faster query speed compared to existing solutions of network name lookups.**

***Index Terms*—Packet Switching, Software Defined Networking, Algorithm design and analysis**

I. Introduction

Significant efforts have been devoted to the investigation and deployment of new network technologies in order to simplify network management and to accommodate emerging network applications. Though different proposals of new network technologies focus on a wide range of issues, one consensus of most new network designs is the separation of network identifiers and locators [27], which are combined in IP addresses in the current Internet. Instead of IP, flat-name or namespace-neutral architectures have been proposed to provide persistent network identifers. A flat or location-independent namespace has no inherent structure and hence imposes no restrictions to referenced elements [5].

The Salter's taxonomy of network elements [27] is one of the early proposals that suggest the separation of network identifiers and locators. We summarize an (incomplete) list of reasons for using flat or location-independent names in proposed network architectures:

- To simplify network management, pure layer-two Ethernet is suggested to interconnect large-scale enterprise and data center networks[18], [12], [31], where MAC addresses are identifiers.
- Software Defined Networking (SDN) uses matching of multiple fields in packet header space to perform fine-grained per-flow control. Flow IDs can also be considered names, though they are not fully flat.
- Flat network identifiers have been suggested by various works to support host mobility and multi-homing, including HIP [23], Layered Naming Architecture [5], and Mobility-First [26].
- AIP [3] applies flexible addressing to ensure trustworthy communication.
- The core network of Long-Term Evolution (LTE) needs to forward downstream traffic according to the Tunnel End Point Identifier (TEID) of the flows [41].

The most critical problem caused by location-independent names is *Forwarding Information Base (FIB) explosion*. A FIB is a data structure, typically a table, that is used to determine the proper forwarding actions for packets, at the data plane of a forwarding device (e.g, switch or router). Forwarding actions include sending a packet to a particular outgoing interface and dropping the packet. Determining proper forwarding actions of the names in a FIB is called name switching. Unlike IP addresses, location-independent names are difficult to aggregate due to the lack of hierarchy and semantics. The increasing population of network hosts results in huge FIBs and their continuing fast growth.

On the other hand, the increasing line speed requires the capability of fast forwarding. To support multiple 10Gb Ethernet links, a FIB may need to perform hundreds of millions of lookups per second. Existing high-end switch fabrics use fast memory, such as TCAM or SRAM, to support intensive FIB query requests. However, as discussed in many studies [36], [9], [37], fast memory is expensive, power-hungry, and hence very limited on forwarding devices. Therefore, achieving *fast queries* with *memory-efficient* FIBs is crucial for the new network architectures that rely on *location";independent names*. If FIBs are small and increase very little with network size, network operators can use relatively inexpensive switches to build large networks and do not need frequent switch upgrades when the network grows. Hence, the cost of network construction and maintenance can be significantly reduced. For

Manuscript received October 24, 2017; revised February 28, 2018; accepted March 9, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. P. C. Lee. The work of Y. Yu and C. Qian was supported by the National Science Foundation under Grant CNS-1701681 and Grant CNS-1717948. The work of Q. Zhang was supported by the NSF under Grant CCF-1525024 and Grant IIS-1633215. A preliminary version of this paper was published in the Proceedings of IEEE ICNP 2017 [38]. (*Corresponding author: Chen Qian.*)

Ye Yu (ye.yu@uky.edu) is with the Department of Computer Science at the University of Kentucky, Lexington, KY, 40508, USA.

Djamal Belazzougui(dbelazzougui@cerist.dz) is with DTISI at CERIST, Ben Aknoun, Alger-Algrie.

Chen Qian (qian@ucsc.edu) is with the Department of Computer Engineering at the University of California Santa Cruz, Stanta Cruz, CA, 95064, USA.

Qin Zhang (qzhangcs@indiana.edu) is with the Computer Science Department at Indiana University Bloomington, Bloomington, IN 47405, USA.

FIB	Construction Time	Query Structure Size (bits)	Query Time	Note
`Concise`	$O(n)$	$\leq 4n \log w$	$O(1)$	Exact 2 memory reads per query.
`(2,4)`-Cuckoo [42]	$O(n)$	$\sim 1.1n(L+\log w)$	$O(1)$	Up to 8 memory reads per query.
SetSep [41]	$O(n \log w)$	$(2+1.5 \log w)n$	$O(\log w)$	No method for updates. Not designed as FIB in [41].
BUFFALO [36]	$O(nt)$	αwn	$O(tw)$	Probabilistic results. Error ratio affected by t and α.
TSS [30]	$O(n(t+\log w))$	$O(n(t+\log w))$	$O(t)$	Designed for names with t fields. $t = O(L)$.

Table I: Comparison among FIBs. n: # of names. L: length of names. w: # of possible actions. **In practice, Concise achieves 7% to 40% memory and >2x speed compared to Cuckoo, though they share the same order of big O time complexity**.

software switches, small FIBs are also important to fit into fast memory such as cache.

In this paper, we present a new FIB design called `Concise`. It has the following properties.

1) Compared to existing FIB designs for name switching, `Concise` supports *much faster name lookup* using *significantly smaller memory*, shown by both theoretical analysis and empirical studies.
2) `Concise` can be efficiently updated to reflect network dynamics. A single CPU core is able to perform millions of network updates per second. `Concise` makes the control plane highly scalable.
3) `Concise` guarantees to return the correct forwarding actions for valid names. It is *not* probabilistic like those using Bloom filters [36], [22].

`Concise` is built on a new network algorithm named Othello Hashing. Othello was inspired by the techniques used in perfect hashing [21], [6]. Different from the static solutions such as Bloomier Filter [7], *our unique contribution on Othello is to utilize the programmable networking techniques to support network dynamics and corresponding updates*. Othello Hashing and `Concise` FIB support fast query and update (addition/deletion of names). In the resource-limited switches (data plane), `Concise` only includes the query component and is optimized for memory efficiency and query speed. The construction and update components are moved to the resource-rich control plane. `Concise` is constructed and updated in the control plane and transmitted to the data plane via a standard API such as OpenFlow. It is the first work to implement minimal perfect hashing schemes to network applications with update functionalities. `Concise` is designed for flat-name lookups. It does *not* support layer-3 longest prefix matching of IP addresses. `Concise` is **a portable solution**, and it can be used in either software or hardware switches. We have implemented `Concise` in three different computing environments: memory mode, CLICK Modular Router [19], and Intel Data Plane Developement Kit [13]. The experiments conducted on an ordinary commodity desktop computer show that `Concise` uses only few MBs of memory to support hundreds of millions lookups per second, when there are millions of names.

The rest of this paper is organized as follows. Sec. II presents related work. We introduce the overview of `Concise` in Sec. III. We present the Othello data structure in Sec. IV and the system design in Sec. V. We then present the system implementation and experimental results in Sec. VI. Sec. VII discusses a few related issues. Finally, we conclude this work in Sec. VIII.

II. Related Work

Location-independent network names. Separating network location from identity has been proposed and kept repeating for over two decades. Numerous network architectures appear in the literature that suggest this concept. As discussed in Sec. I, a number of new network architectures adopt location-independent names. A location-independent name can be a MAC address, a tuple consisting of several packet header fields [17], a file name [14], [40], a TEID [41], etc. To route packets for flat names, ROFL [8] and Disco [29] propose to use compact routing to achieve scalability and low routing stretch. ROME [25] is a routing protocol for layer-two networks that uses greedy routing whose routing table size is independent of network size. `Concise` is a forwarding structure and does not deal with routing.

FIB scalability. We name some techniques used for FIBs and compare them in Table I.

Hashing is a typical approach to reduce the memory cost of FIBs for name-based switching. CuckooSwitch [42] uses carefully revised Cuckoo hash tables [24] to reach desirable performance on specific high-end hardware platforms. ScaleBricks [41] also makes use of a memory-efficient data structure *SetSep* to partition a FIB to different nodes in a cluster, it does not store the names as well. We provide a comprehensive comparison of Cuckoo hashing, and `Concise` in Sec. VII-B. The use of Bloom filters has been proposed in some designs such as BUFFALO [36], [22]. However, they may forward packets incorrectly due to the false positives in Bloom filters, causing forwarding loops and bandwidth waste. For IP lookups, SAIL [35] and Portire [4] demonstrate desirable throughput for IPv4 FIB queries. These solutions are usually based on hierarchical tree structures, and their performance are challenged by FIBs with large number of flat names. The Tuple Space Search algorithm (TSS) [30] is widely used for name matching with multiple files, such as in OpenVswitch and PIECES [28]. It is not designed for flat-name switching. Other solutions use hardware to accelerate name switching. For example, Wang *et al.* [32] uses GPU to accelerate name lookup in Named Data Networks. A recent work utilize Bloom filters for set queries, which cannot be applied to our situation [33]. A recent work Difference Bloom Filter (DBF) [34] provides more more accurate query results than Bloom filters with faster query speed but still has false positives.

Minimal perfect hashing. The data structure used in this work, Othello, is built upon the studies on minimal perfect hashing. In particular, MWHC [21] is able to generate order-preserving minimal perfect hash functions using a random

hypergraph. MWHC is also presented as Bloomier Filter in [7]. The differences between Othello and these studies include: (1) Othello uses a bipartite graph instead of a general random hypergraph. This design allows much simpler concurrency control mechanism. (2) The original researches on MWHC and Bloomier Filter are designed for static scenarios. The dynamic update mechanism was not presented, which could be more complicated in practice. (3) Othello is optimized for real network conditions. It performs different functionalities on the control plane and the data plane. Othello aims to support fast flat name switching, while MWHC is for finding minimum perfect hash functions [21] and Bloomier Filter is designed for approximate evaluation queries [7].

III. Design Overview

Consider a network of n hosts identified by unique names. The hosts are connected by SDN-enabled switches. As shown in Figure 1. A logically central controller is responsible of deciding the routing paths of packets. Each switch includes a FIB. The controller communicates with each switch to install and update the FIB

Each packet header includes the name of the destination host, denoted as k. Upon receiving a packet, the switch decides the forwarding action of the packet, such as forward to a port or drop. We assume the controller knows the set S of all names in the network. In addition, `Concise` only accepts queries of valid names, i.e., $k \in S$. We assume that firewalls or similar network functions are installed at ingress switches to filter packets whose destination names do not exist. More discussion about eliminating invalid names is presented in Sec. VII-A.

`Concise` makes use of a data structure named Othello. Othello exists in both the switches (data plane) and the controller (control plane). It has two different structures in the data plane and control plane:

- **Othello query structure** implemented in a switch is the FIB. It only performs name queries. The memory efficiency and query speed is optimized and the update component is removed.
- **Othello control structure** implemented in the controller maintains the FIB as well as other information used for FIB construction and updates, such as the routing information base (RIB).

Upon network dynamics, the control structure computes the updated FIBs of the affected switches. The modification is then sent from the controller to each switch.

Separating the query and control structures is a perfect match to the programmable networks such as SDN. We call this new data structure design as a **Polymorphic Data Structure** (PDS). PDS is the key reason that we can apply minimal perfect hashing techniques in programable networks. PDS differs from the current SDN model. SDN separates the RIB and FIB to the control and data plane respectively. We further move part of the FIB to the control plane to minimize the data plane resource cost.

IV. Othello Hashing

In this section, we describe the Othello data structure. Inspired by the MWHC minimal perfect hashing algorithm

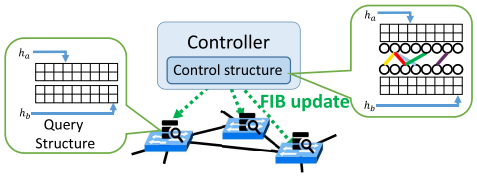


Figure 1: Network Overview of `Concise`

[21], we design Othello specially for maintaining the FIB. The Bloomier filter [7] can be considered as a special case of the static version of Othello.

The basic function of a FIB is to classify all names into multiple sets, each of which represents a forwarding action. Let S be the set of all names. $n = |S|$. An Othello classifies n names into two disjoint sets X and Y: $X \cup Y = S$ and $X \cap Y = \varnothing$. Othello can be extended to classify names into d ($d > 2$) disjoint sets, serving as a FIB with d actions.

A. Definitions

An Othello is a seven-tuple $\langle m_a, m_b, h_a, h_b, \boldsymbol{a}, \boldsymbol{b}, G\rangle$, defined as follows.

- Integers m_a and m_b, describing the size of Othello.
- A pair of uniform random hash functions $\langle h_a, h_b\rangle$, mapping names to integer values $\{0, 1, \cdots, m_a-1\}$ and $\{0, 1, \cdots, m_b-1\}$, respectively.
- Bitmaps $\boldsymbol{a}$ and $\boldsymbol{b}$. The lengths are m_a and m_b respectively.
- A bipartite graph G. During Othello construction and update, G is used to determine the values in $\boldsymbol{a}$ and $\boldsymbol{b}$.

Figure 2 shows an Othello example. We require that $m_a = \Theta(n)$, $m_b = \Theta(n)$, and $m_a m_b > n^2$. We provide two options to determine the values m_a and m_b. 1) m_a is the smallest power of 2 such that $m_a \geq 1.33n$ and $m_b = m_a$. 2) m_a is the smallest power of 2 such that $m_a \geq 1.33n$ and m_b is the smallest power of 2 such that $m_b \geq n$. A user may choose either option. The difference is that for Option 1 we establish a rigorous proof of constant update time and for Option 2 we establish the proof with a constraint on n. However Option 2 provides slightly better empirical results.

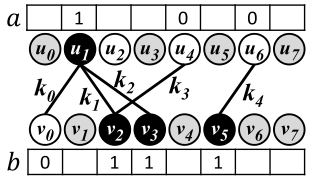
Othello supports a query operation as follows. For a name k, it computes $\tau(k) \in \{0, 1\}$. If $k \in X$, $\tau(k) = 0$. If $k \in Y$, $\tau(k) = 1$. If $k \notin S$, $\tau(k)$ returns 0 or 1 arbitrarily. The values of $\boldsymbol{a}$ and $\boldsymbol{b}$ are determined during Othello construction, so that $\tau(k)$ can be computed by:

$$\tau(k) = \boldsymbol{a}[h_a(k)] \oplus \boldsymbol{b}[h_b(k)]$$

Here, $\oplus$ is the *exclusive or* (`XOR`) operation. In other words, if $k \in X$, $\boldsymbol{a}[h_a(k)] = \boldsymbol{b}[h_b(k)]$; if $k \in Y$, $\boldsymbol{a}[h_a(k)] \neq \boldsymbol{b}[h_b(k)]$.

B. Othello Operations

Othello is maintained via the following operations.



k	$h_a(k)$	$h_b(k)$	$\tau(k)$
k_0	1	0	1
k_1	1	2	0
k_2	1	3	0
k_3	4	2	1
k_4	6	5	1

Figure 2: Example of Othello of $n = 5$ names with $m_a = m_b = 8$. Left: Bipartite graph G and bitmaps $\boldsymbol{a}$ and $\boldsymbol{b}$. Right: five names $k_0, k_3, k_4 \in X$ and $k_1, k_2 \in Y$; the hash values and $\tau(k)$ values.

- `construct`(X, Y): Construct an Othello for two name sets X and Y.
- `addX`(k) and `addY`(k): add a new name k into the set X or Y.
- `alter`(k): For a name $k \in X \cup Y$, move k from set X to Y or from Y to X. After this operation, the query result $\tau(k)$ is changed.
- `delete`(k): For a name $k \in X \cup Y$, remove k from set X or Y.

*1) **Construction**:* The `construct` operation takes as input two sets of names X and Y. The output is an Othello $\boldsymbol{O} = \langle m_a, m_b, h_a, h_b, \boldsymbol{a}, \boldsymbol{b}, G\rangle$.

Here, G is used to determine the hash function pair and the values of $\boldsymbol{a}$ and $\boldsymbol{b}$. $G = (U, V, E)$. $|U| = m_a$, $|V| = m_b$. A vertex $u_i \in U$ or $v_j \in V$ corresponds to bit $\boldsymbol{a}[i]$ or $\boldsymbol{b}[j]$. Each edge in E represents a name. There is an edge $(u_i, v_j) \in E$ if and only if there is a name $k \in S$ such that $h_a(k) = i$ and $h_b(k) = j$.

For each vertex that is associated with at least one edge, the corresponding bit is set to 0 or 1. A vertex associated with bit 0 is colored in white and a vertex associated with bit 1 is colored in black. For vertices that have no associated edges, the value of the corresponding bits can be set to 0 or 1 arbitrarily, because they do not affect any $\tau(k)$ value for $k \in S$. In order to assign correct values of $\boldsymbol{a}$ and $\boldsymbol{b}$, Othello requires G to be *acyclic*.

The construction algorithm consists of two phases.

• *Phase I: Selecting the hash function pair.*

In this phase, Othello finds a hash function pair $\langle h_a, h_b\rangle$. We assume there are many candidate hash functions and will discuss the implementation in Sec. V-B. In each round, two hash functions are chosen randomly and G is accordingly generated. We use Depth-First-Search (DFS) on G to test whether it includes a cycle, which takes $O(n)$ time. The order in which the edges are visited during the DFS, i.e, the DFS order of the edges is recorded to prepare for the second phase. Note that if two or more names generate edges with the same two endpoints, we will consider as if there is a cycle. If G is cyclic, the algorithm will select another pair of hash functions until an acyclic G is found.

• *Phase II: Computing the bitmaps.*

In this phase, we assign values for the two bitmaps $\boldsymbol{a}$ and $\boldsymbol{b}$. First, the values in $\boldsymbol{a}$ and $\boldsymbol{b}$ are marked as undefined. Then, we execute the followings for each $e = (u_i, v_j)$ in the DFS order of the edges: Let k be the name that generates e. If none of $\boldsymbol{a}[i]$ and $\boldsymbol{b}[j]$ has been assigned, let $\boldsymbol{a}[i] \leftarrow 0$ and $\boldsymbol{b}[j] \leftarrow \tau(k)$. If there is only one of $\boldsymbol{a}[i]$ and $\boldsymbol{b}[j]$ has been assigned, we can always assign an appropriate value to the other one, such that $\boldsymbol{a}[i] \oplus \boldsymbol{b}[j] = \tau(k)$. As G is acyclic, following the DFS order, we will never see an edge such that both $\boldsymbol{a}[i]$ and $\boldsymbol{b}[j]$ have values.

We show the pseudocode of Othello construction in Algorithm 1.

Input: Key-set X,Y.
Output: An Othello structure $\langle m, h_a, h_b, a, b, G\rangle$
begin
1 $S \leftarrow X \cup Y$.
2 select m value according to $n = |S|$.
`/* Phase I: decide hash function pair */`
3 **repeat**
4 Randomly select hash function h_a, h_b.
until `GeneratedGraphIsAcyclic`(S, h_a, h_b).
`/* Phase II: Compute bitmaps */`
5 Compute $G = (U, V, E)$ using h_a, h_b and S.
6 Execute Depth-First-Search on G.
7 $(e_1, e_2, \cdots, e_n) \leftarrow$ the DFS order of E.
8 Mark all $\boldsymbol{a}[i], \boldsymbol{b}[j] (0 \le i, j < m)$ as `unassigned`.
9 **for** $t = 1, 2, \cdots, n$ **do**
10 $k \leftarrow$ the corresponding name for e_t.
11 **if** $k \in X$ **then** $v \leftarrow 0$ **else** $v \leftarrow 1$.
12 $i \leftarrow h_a(k)$; $j \leftarrow h_b(k)$.
13 **if** both $\boldsymbol{a}[i]$ and $\boldsymbol{b}[j]$ are `unassigned` **then**
14 $\boldsymbol{a}[i] \leftarrow 0$; $\boldsymbol{b}[j] \leftarrow v$.
15 **else if** $\boldsymbol{a}[i]$ is `unassigned` **then**
16 $\boldsymbol{a}[i] \leftarrow \boldsymbol{b}[j] \oplus v$.
17 **else** `/* b[j] isunassigned */`
18 $\boldsymbol{b}[j] \leftarrow \boldsymbol{a}[i] \oplus v$.
end
end
end

Algorithm 1: Othello `construct` procedure

Note that the edges of G are only determined by $S = X \cup Y$ and the hash function pair $\langle h_a, h_b\rangle$. If we find G to be cyclic for a given S and a pair $\langle h_a, h_b\rangle$, we shall use another pair $\langle h_a, h_b\rangle$ to make G acyclic. We show that for a randomly selected pair of hash functions $\langle h_a, h_b\rangle$, the probability of G to be acyclic is very high:

Theorem 1. *Given set of names $S = X \cup Y$, $n = |S|$. Suppose h_a, h_b are randomly selected from a family of fully random hash functions. $h_a : S \to \{0, 1, \cdots, m_a - 1\}$, $h_b : S \to \{0, 1, \cdots, m_b - 1\}$. Then the generated bipartite graph G is acyclic with probability $\sqrt{1 - c^2}$ when $n \to \infty$, where $c = \frac{n}{\sqrt{m_a m_b}}$, $c < 1$.*

Proof. Let $G = (U, V, E)$ be a bipartite random graph with $|U| = m_a$, $|V| = m_b$, $|E| = n$, where each edge is independently taken at random with probability $\frac{n}{m_a m_b}$. Let $\mathcal{C}_{2\ell}$ be the set of cycles of length 2ℓ ($\ell \ge 1$) in the complete bipartite graph K_{m_a, m_b}.

As proved in [20, Theorem 1], the number of cycles of any even length in G, represented as a random variable $\mathcal{X}$, converges to a Poisson distribution with parameter λ_e, where

$$\lambda_e = -\frac{1}{2}\ln(1 - c^2).$$

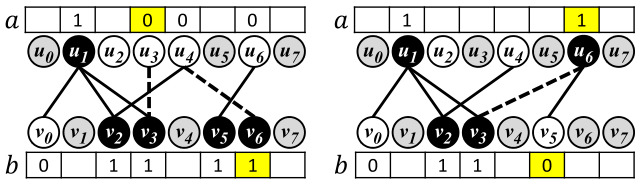


Figure 3: Example of adding names into Othello. Dashed edges: added keys. Highlighted cells: modified values in a and b. Left: e adds isoloated nodes to existing connected components, Right: e joins two existing non-trivial connected componens.

Therefore, the probability that G contains no cycle is

$$\Pr(\mathcal{X} = 0) = e^{-\lambda_e} = \sqrt{1 - c^2}.$$

□

When G is acyclic, we say that $\langle h_a, h_b \rangle$ is a *valid hash function pair* for S.

When $c \leq 0.75$ (i.e, $n \leq 0.75m$), $\sqrt{1-c^2} \geq 0.66$. Hence the expected number of rounds to find an acyclic G in Phase I is $\frac{1}{\sqrt{1-c^2}} \leq 1.51$ when $c < 0.75$. The time complexity is $O(n)$ in each round. The second phase takes $O(n)$ time to visit n edges and assign values of $\boldsymbol{a}$ and $\boldsymbol{b}$. Hence, the total expected time of `construct` is $O(n)$.

*2) **Name addition**:* To `add` a name k to X or Y, the graph G and two bitmaps should be changed in order to maintain the correct result $\tau(k)$.

The algorithm first computes the edge $e = (u, v)$ to be added to G for k, $u = u_{h_a(x)}$, $v = v_{h_b(x)}$. Note that G can be decomposed into connected components. e must fall in one of the following cases:

• *Case I:* u and v belong to the same connected component `cc`. Adding e to G will introduce a cycle. In this case, we have to re-select a hash function pair $\langle h_a, h_b \rangle$ until a valid hash function pair is found for the new name set $S \cup \{k\}$. The `construct` algorithm is used to perform this process.

• *Case II:* u and v are in two different connected components. As shown in Figure 3, the edge e either (1) adds an isolated node to a connected component in G; (2) joins two connected components in G. Combining the two connected components and the new edge, we have a single connected component that is still acyclic. As discussed in Sec. IV-B1, it is simple to find a valid coloring plan for an acyclic connected component. Hence, the values of $\boldsymbol{a}$ and $\boldsymbol{b}$ can also be set properly. In fact, at least one of the two connected components can keep the existing value assignments.

Complexity Analysis. We now compute the time complexity of `add` using three theorems. In particular, we will show that the time complexity of the `add` operation is $O(1)$. The important parameter that governs the complexity of an insertion is the *susceptibility* of the graph G, which is defined as the expected size of the connected component that contains a randomly chosen node, and is denoted by $\chi(G)$.

We give a closed-form estimation for $\chi(G) = \frac{1}{1-p}$ where $p = \frac{n(m_a+m_b)}{2m_a m_b}$, and prove that $\chi(G)$ has a constant upper-bound $E[\chi(G)] \leq 4$. As stated before, there are two options in choosing values m_a and m_b. In Option 1, $m_a = m_b$ and in Option 2, $m_a = m_b$ or $m_a = 2m_b$. We are able to compute the closed-form formulae for $\chi(G)$ when $m_a = m_b$. For the case $m_a = 2m_b$, we give a looser upper bound. The numerical estimation shows that the upper bound $E[\chi(G)] \leq 4$ is true for both of the two situations where $m_a = m_b$ and $m_a = 2m_b$.

For the sake of analysis we let $\mathcal{G}_A(m_a, m_b, n)$ be a random acyclic graph generated using the same process as $\mathcal{G}(m_a, m_b, n)$ except that an edge is not added if it introduces a cycle in the graph. It could also be generated by repeatedly generating graphs $\mathcal{G}(m_a, m_b, n)$ until we get an acyclic graph. It is evident that this random graph model corresponds to the graphs constructed and maintained by Othello.

For the case $m_a = m_b$ we show Theorem 2. For the case $m_a = 2m_b$ we show Theorem 3. Theorem 4 concludes that the time complexity of `add` is $O(1)$.

Theorem 2. *Suppose we have a random graph $\mathcal{G}_A(m_a, m_b, n)$ where $m_a = m_b$ and we randomly select a node w in $\mathcal{G}_A$. Let $\texttt{cc}(w)$ be the connected component containing w. Then the expected value of $|\texttt{cc}(w)|$ is $\frac{m_a}{m_a - n}$ as $n \to \infty$.*

Proof. Let $\chi(G) = E[|\texttt{cc}(w)|]$ where w is randomly selected from G and $|\texttt{cc}(w)|$ denotes the number of nodes in $\texttt{cc}(w)$. In [10, Lemma 1], it was proved that for a random sparse graph $\mathcal{G}(m_a, m_a, n)$ with n edges, we have $\chi(G) = \frac{2m_a}{2m_a - 2n}$ when $n \to \infty$ given that $n < 0.999m_a$. We will show that the same bound holds for a graph $\mathcal{G}_A(m_a, m_a, n)$. It is well known that the largest connected component in a random graph with n edges and m nodes with $n \leq 0.99 \cdot m/2$ has size $O(\log n)$ with probability $1 - \frac{1}{n^{10}}$ [10].

We now generate a graph $\mathcal{G}_A(m_a, m_a, n)$ by generating the edges one by one. If an edge (v, w) makes the graph cyclic, then we do not add it, but instead put it into a set S. Let E be the set of n edges in the generated acyclic graph G_1. Then graph G_2 with the set of edges $E \cup S$ will clearly be a graph $\mathcal{G}(m_a, m_a, n')$ with $n' = n + O(\log^2 n) \leq 0.999m/2$. Now we have that $\chi(G_1) \leq \chi(G_2)$ and $\chi(G_2) = \frac{2m_a}{2m_a - 2n'} \to \frac{2m_a}{2m_a - 2n}$ when $n \to \infty$. □

Theorem 3. *For a random graph $\mathcal{G}_A(m_a, m_b, n)$ where $m_a = 2m_b$, $n \leq 0.65m_b$, and randomly select a node w in $\mathcal{G}$. Let $\texttt{cc}(w)$ be the connected component containing w. Then the expected value of $|\texttt{cc}(w)|$ is $O(1)$.*

Proof. Again let $\chi(G) = E[|\texttt{cc}(w)|]$ where w is randomly selected from G. We generate a graph $\mathcal{G}_A(m_a, m_b, n)$ with $n \leq 0.65m_b$ as follows. Let V_a with $|V_a| = m_a$ be the set of nodes on the left side, and V_b with $|V_b| = m_b$ be the set of nodes on the right side. We generate edges one by one from random graph $\mathcal{G}_A(m_a + m_b, n)$, and reject an edge (v, w) if either $(v \in V_a \wedge w \in V_a)$ or $(v \in V_b \wedge w \in V_b)$. The probability of accepting an edge is thus $\frac{4}{9}$. We stop the generation when we have finished generating the n edges, and we denote the resulting graph by G_1. We let G_2 be the graph obtained by adding all the rejected edges back to G_1. It is clear that $\chi(G_1) \leq \chi(G_2)$. Moreover, G_2 is a random graph $\mathcal{G}_A(m_a + m_b, n')$ with $n' = \frac{9}{4}n \pm O(\sqrt{n})$ with probability

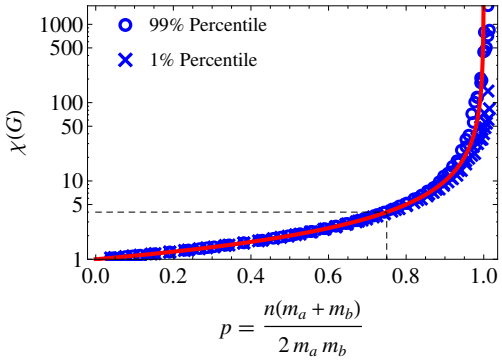


Figure 4: $\chi(G)$ of acyclic graphs vs parameter p. Red curve: $\frac{1}{1-p}$

$1 - \frac{1}{n^{10}}$. According to Theorem 3.3(i) in [16], for a graph $G_3 = \mathcal{G}(m_a + m_b, n')$:

$$\begin{aligned}\chi(G_3) &\leq \frac{m_a + m_b}{m_a + m_b - 2n'} = \frac{3m_b}{3m_b - 2 \cdot \frac{9}{4}n} \\ &\leq \frac{3\frac{n}{0.65}}{3\frac{n}{0.65} - 2 \cdot \frac{9}{4}n} = O(1).\end{aligned}$$

We can use the same argument as in the proof of Theorem 2 to show that the susceptibility for a graph $\mathcal{G}_A(m_a + m_b, n')$ is the same as for a graph $\mathcal{G}(m_a + m_b, n')$ which concludes the proof. □

Theorem 4. *Assuming h_a, h_b are randomly selected from a family of fully random hash functions, an insertion into an Othello with n existing names will take constant amortized expected time when $m_a = m_b$, or when $m_a = 2m_b$ and $n \leq 0.65m_b$.*

Proof. In the algorithm described in Section. IV-B2, during an insertion, we have to add an edge that connects a randomly selected node $u \in U$ to another randomly selected node $v \in V$. We will first bound the amortized expected cost of insertions that fall in *Case I* and then the induced cost of insertions that fall in *Case II*. Let $|\texttt{cc}(w)|$ be the size of the connected component that contains node w. Let $|\texttt{cc}_b(w)|$ be the number of nodes in $\texttt{cc} \cup V$. $|\texttt{cc}_b(w)| < |\texttt{cc}(w)|$.

The probability that node v falls in the same connected component as node w is $\frac{|\texttt{cc}_b(w)|}{m_b} \leq \frac{|\texttt{cc}(w)|}{m_b}$ which is the probability of reconstruction. Since the reconstruction takes expected $O(n)$ time, the amortized time cost for reconstruction during the insertion of one name is $\frac{|\texttt{cc}_b(w)|}{m_a} \cdot O(n) = O(|\texttt{cc}(w)|) = O(1)$.

For *Case II*, the cost is clearly $O(|\texttt{cc}(w)|+|\texttt{cc}(v)|) = O(1)$, since we have to traverse the connected component that results from merging the two connected components that contain w and v. □

Note we have a rigorous proof for Option 1 but Option 2 provides slightly better empirical results. It is reasonable to conjecture that Theorem 4 also holds for $m_a = 2m_b$ without the constraint $n \leq 0.65m_b$.

Numerical estimation of $\chi(G)$: We conjecture that $\frac{1}{1-p}$, where $p = \frac{n(m_a+m_b)}{2m_a m_b}$ is a good estimation for $\chi(G) = E[|\texttt{cc}(w)|]$, and present numerical simulation to support our conjecture. We generate acyclic bipartite graphs with random m_a, m_b, and n values (within the range 10K $\sim$ 1M). Then we compute their $\chi(G)$ value. For a particular $p = \frac{n(m_a+m_b)}{2m_a m_b}$ value, we randomly sample at least 500 graphs with different m_a,m_b, and n. In Figure 4, we plot the 1-th and 99-th percentile of $\chi(G)$.

As shown in Figure 4, when p is not so close to 1, the sampled $\chi(G)$ values are very close to $\frac{1}{1-p}$. When p grows larger, the sampled $\chi(G)$ values tend to grow slower than $\frac{1}{1-p}$. Hence we conclude that $\frac{1}{1-p}$ is a good upper bound for $\chi(G)$. In Othello, $\frac{4}{3}n \leq m_a < \frac{8}{3}n$, $n \leq m_b < 2n$. m_a and m_b must be powers of 2. For this choice of parameters we can see that $p = \frac{n(m_a+m_b)}{2m_a m_b} \leq 0.75$ and so $\chi(G) \leq 4$ which is a small constant.

This estimated value $\chi(G) = \frac{1}{1-p}$ is in coherence with the evaluation results on `Concise` updates shown in Figure 11.

Othello size growth. After adding a name into Othello, $n = |S|$ grows and may violate $m_a \geq 1.33n$ and $m_b \geq n$. $\chi(G)$ also grows. Hence, Othello may choose to reconstruct the graph G in order to guarantee very low amortized time overhead of future operations. However, Othello works correctly as long as G is acyclic, even when $m_a < 1.33n$ or $m_b < n$. Hence, Othello does not deal with the requirement on m_a and m_b explicitly for additions. Although the $\chi(G)$ value may grow as more names are added to Othello, it is always smaller than 10 in our experiments. The expected time to add a name to Othello is still $O(1)$ in practice.

When adding a new name falling in Case I, the values of m_a and m_b will be updated by `construct`, which guarantees $m_a \geq 1.33n$ and $m_b \geq n$.

*3) **Set change for a name**:* Operation `alter`(k) is used to move a name k from set X to set Y (or from Y to X). The bitmaps $\boldsymbol{a}$ and $\boldsymbol{b}$ should be modified so that $\tau(k)$ is changed from 0 to 1 (or from 1 to 0). The graph G does not change during `alter`(k). We only need to change the coloring plan of the connected component that contains the edge $e = (u_{h_a(k)}, v_{h_b(k)})$. One approach is to "flip" the colors of all vertices at one side of e, i.e., to change 0 to 1, and to change 1 to 0. The amortized time cost is $O(1)$.

*4) **Name deletion**:* `delete`(k) can be done by simply removing the edge $(u_{h_a(k)}, v_{h_b(k)})$ from G. The bitmaps $\boldsymbol{a}$ and $\boldsymbol{b}$ are not modified because the values of $\tau(k)$ after deleting k do not matter anymore. The time complexity is $O(1)$.

C. Query structure and control structure

Each Othello is a seven-tuple $\langle m_a, m_b, h_a, h_b, \boldsymbol{a}, \boldsymbol{b}, G\rangle$. Note that for a query on Othello, only the first six elements are necessary for computing the τ value. The information stored in G is not needed for the query operation. Hence, we let the switches only maintain the six-tuple $\langle m_a, m_b, h_a, h_b, \boldsymbol{a}, \boldsymbol{b}\rangle$ in their local memory, namely the *Query structure*. Storing this six-tuple takes $2m + O(1)$ bits of memory space. The time cost for each query of Othello is equal to the sum of the cost

of computing two hash values, two memory accesses for the two bitmaps, and one XOR arithmetic operation.

In comparison, the network controller maintains the seven-tuple, namely the *Control Structure*. The controller is responsible for maintaining the FIB of the switches in the network. The switches execute the queries on the query structures.

D. Summary of Othello Properties

An Othello is decomposed into a query structure running in the data plane and a control structure in the control plane. The query structure uses $\leq 4n$ bits for n names. Every query takes a small constant time including computing two hash values and two memory accesses. The control structure uses $O(n)$ bits. The expect time complexity is $O(n)$ for construction and $O(1)$ for name addition, deletion, and set change. Note that *the distribution of names in X and Y has no impact on the space and time cost of Othello*, because G only depends on S and $\langle h_a, h_b \rangle$. In Sec. V-A, we demonstrate the extension of Othello. It classifies names into $d > 2$ disjoint sets, while still requiring small memory and constant query time.

V. System Design of Concise

We present how to build Concise using the Othello data structure as follows. The design also includes the implementation details of FIB update and concurrency control.

A. Extension of Othello for Network Lookups

The extension of Othello to support classification for more than two sets is called a Parallel Othello Group (POG). An l-POG is able to classify names into 2^l disjoint sets. It serves as a FIB with 2^l forwarding actions. Let $Z_0, Z_1, \cdots, Z_{2^l-1}$ be the 2^l disjoint sets of names. Let $S = Z_0 \cup Z_1 \cup \cdots \cup Z_{2^l-1}$. A query on the l-POG for a name $k \in S$ returns an l-bit integer $\tau(k)$, indicating the index of the set that contains k, i.e., $k \in Z_{\tau(k)}$.

The idea of POG is as follows. Consider l Othellos $\boldsymbol{O}_1, \boldsymbol{O}_2, ..., \boldsymbol{O}_l$. Each $\boldsymbol{O}_i$ classifies keys in set X_i and Y_i ($1 \leq i \leq l$), where X_i and Y_i satisfies:

$$X_i = \bigcup_{(j \bmod 2^i) < 2^{i-1}} Z_j; \qquad Y_i = \bigcup_{(j \bmod 2^i) \geq 2^{i-1}} Z_j.$$

Let $\tau_i(k)$ be the query result of $\boldsymbol{O}_i$ for name k. Consider the l-bit integer $((\tau_l(k)\tau_{l-1}(k)\cdots\tau_1(k))_2$. Note that $\tau_i(k) = 0$ if and only if $k \in X_i$. Meanwhile, $Z_{\tau(k)} \subset X_i$ if and only if $(\tau(k) \mod 2^i) < 2^{i-1}$ (the i-th least significant bit of $\tau(k)$ is 0). Hence, the i-th least significant bit of $\tau(k)$ equals to $\tau_i(k)$. i.e,

$$\tau(k) = ((\tau_l(k)\tau_{l-1}(k)\cdots\tau_1(k))_2$$

For each i ($1 \leq i \leq l$), $X_i \cup Y_i = S$. i.e., the l Othellos share the same S. Recall that the edges in G is determined by only $S = X \cup Y$ and $\langle h_a, h_b \rangle$, and $\langle h_a, h_b \rangle$ is decided during construct by S. The l Othellos may share the same $\langle h_a, h_b \rangle$ and same edges in G. However, the bitmaps in different Othellos are different.

Parallelized execution with bit slicing. Each operation of an l-POG consists of operations on the l Othellos. Using the bit slicing technique, these operations can be executed in parallel. The bit slicing technique is widely used to group executions in parallel [2]. An l-POG query structure includes l, m, h_a, h_b and two vectors A and B. Each of A and B contains m l-bit integers. Consider all the i-th bits of the elements in A. These bits can be viewed as a *slice* of the array A. The i-th slice of A is used to represent bitmap $\boldsymbol{a}_i$. The slices of B are defined similarly. Using this technique, $\tau(k)$ can be computed using one arithmetic operation by:

$$\tau(k) = A[h_a(k)] \oplus B[h_b(k)]$$

When l is not larger than the word size of the platform, each l-POG query only requires two memory accesses for fetching $A[i]$ and $B[j]$. The arithmetic operation includes computing the hash functions and the XOR.

All Othello operations can be decomposed into two steps: (1) modifications on G, (2) operations on some bits in $\boldsymbol{a}$ and $\boldsymbol{b}$. In an l-POG, the l Othellos share the same G and *the first step is only executed once* for *all* l Othellos. Hence the bit slicing technique also applies to all other operations of POG.

Therefore, the expected time cost of each name addition, deletion, or set change operation is only $O(1)$, instead of $O(l)$. The time complexity of POG construction is still $O(n)$.

B. Selection of Hash functions

The hash function pair is critical for system efficiency. Ideally, h_a and h_b should be chosen from a family of fully random and uniform hash functions. Similar to the implementation of CuckooSwitch [42], we apply a function $H(k, \texttt{seed})$ to generate the hashes in our implementation. Here, H is a particular hashing method and seed is a 32-bit integer. We let $h_a(k) = H(k, \texttt{seed}_a)$ and $h_b(k) = H(k, \texttt{seed}_b)$. Thus, $\langle h_a, h_b \rangle$ is uniquely determined by a pair of integers $\langle \texttt{seed}_a, \texttt{seed}_b \rangle$.

The proper hashing method $H()$ is platform-dependent. Concise uses the CRC32c function for robust and faster hash results, which is then effectively mapped to a t-bit integer value where $m_a = 2^t$ or $m_b = 2^t$. Evaluation shows that CRC32c demonstrates desirable performance in practice.

C. FIB Update and Concurrency Control

We assume that there is one logically centralized controller in the network. Upon network dynamics, the controller computes the POGs for a number of switches and update the query structures in the switches by FIB update messages using a standard SDN API. If m, h_a, h_b do not change during the update, an update message only contains a list of elements to be modified in A and B. Otherwise, it contains the full query structure of l-POG $\langle m, h_a, h_b, A, B \rangle$.

After receiving a FIB update message, a Concise switch modifies its POG query structure. Instead of locks, Concise uses simple bit vectors to prevent read-write conflicts in the query structure. Experimental results show that the concurrency control mechanism has a negligible impact on the network performance.

While each POG query is computed using two elements in A and B, there is a chance of a read-write conflict during the

Data: New value at some indexes in A and B: $A[i_1], A[i_2], \cdots, B[j_1], B[j_2], \cdots$.
Result: Updated Concise query structure

Affected $\leftarrow \emptyset$;
foreach $i \in \{i_1, i_2, \cdots\}$ **do**
  Affected $\leftarrow$ *Affected* $\cup\{i \mod 512\}$
end
foreach $i \in$ *Affected* **do**
  $D_1[i] \leftarrow 1 \oplus D_1[i]$
end
// reorder barrier
Update $A[i_1], A[i_2], \cdots, B[j_1], B[j_2], \cdots$;
// reorder barrier
foreach $i \in$ *Affected* **do**
  $D_2[i] \leftarrow 1 \oplus D_2[i]$
end

Algorithm 2: Update procedure for Concise

Data: Concise query structure and name k
Result: Query result $\tau(k)$

$i \leftarrow h_a(k)$;
$j \leftarrow h_b(k)$;
$p \leftarrow i \mod 512$;
while *true* **do**
  $\boldsymbol{\delta_2} \leftarrow D_2[p]$;
  // reorder barrier
  $\alpha \leftarrow A[i]$;
  $\beta \leftarrow B[j]$;
  // reorder barrier
  $\boldsymbol{\delta_1} \leftarrow D_1[p]$;
  if $\boldsymbol{\delta_2} = \boldsymbol{\delta_1}$ **then**
    return $\alpha \oplus \beta$
  end
end

Algorithm 3: Query procedure on Concise

update. In Concise, the query always returns correct result. Such concurrency issue is addressed as follows.

Concurrency requirements. Let A, B be the two vectors of the query structure before an update and A', B' be the ones after the update. For a name k that exists in the FIB before and after the update, suppose $i = h_a(k)$ and $j = h_b(k)$. Both $A[i] \oplus B[j]$ and $A'[i] \oplus B'[j]$ are considered as correct actions, although they may be different. Note that, when $A[i] = A'[i]$, the values $A'[i] \oplus B[j]$ and $A[i] \oplus B'[j]$ are both correct query results, no matter how read/write events are ordered. Inconsistency only happens when both $A[i]$ and $B[j]$ are changed during the update.

Concurrency control design.

Concise observes whether the vector A is being modified. For a query for name k, if an update that affects $A[i]$ is being executed, Concise does not execute the query until the update finishes. Concise maintains two bit vectors D_1 and D_2 for concurrency control. All bits in D_1 and D_2 are set to 0 during the initialization. Each index i ($0 \leq i < m$) corresponds to an index $p(i)$ in D_1 and D_2. The lengths of D_1 and D_2 are set to 512 bits and $p(i) = i \mod 512$.

Update procedure. A pseudocode of the update procedure is described in Algorithm 2. Before an update of the POG that will change some elements of A, Concise flips the corresponding bits in D_1, i.e., change 0s to 1s and 1s to 0s. After the update, it flips the bits with same indexes in D_2. For any index i, when Concise observes $D_1[p(i)] \neq D_2[p(i)]$, there must be no ongoing update that affects $A[i]$. Note that even if a bit index corresponds to multiple elements that are changed in an update, the bit is only flipped once.

Query procedure. A pseudocode of the query procedure is described in Algorithm 3 The query procedure for name k includes the following three steps. (1) Fetch the bit $\boldsymbol{\delta_2} = D_2[p(i)]$. (2) Fetch the value of $A[i]$ and $B[j]$. (3) Fetch $\boldsymbol{\delta_1} = D_1[p(i)]$. If $\boldsymbol{\delta_2} = \boldsymbol{\delta_1}$, compute $A[i] \oplus B[j]$ and return it as the query result. Otherwise, $\boldsymbol{\delta_2} \neq \boldsymbol{\delta_1}$ and we know that the POG is currently being updated and the update affects $A[i]$. The query for k will stop and is put in a later place of the query event queue. Concise uses reordering barrier instructions to ensure the execution order in both update and query procedures.

Here, the order of flipping $D_1[p(i)]$ and $D_2[p(i)]$ during an update and the order of getting their values during a query are different. Any updates that affect $A[i]$ and start during a query must result in $\boldsymbol{\delta_2} \neq \boldsymbol{\delta_1}$.

The above procedures of update and query should be executed in the given explicit order. This can be specified by compiler reorder barriers on strong memory model platforms such as x86_64, or fence instructions on weak memory model platforms such as ARM.

VI. Implementation and Evaluation

We implement Concise on three platforms and conduct extensive experiments to evaluate its performance.

A. Implementation Platforms

1. Memory-mode. We implement the POG query and control structures, running on different cores of a desktop computer. In addition, we use a discrete-event simulator to simulate other data plane functions such as queuing. The memory-mode experiments are used to compare the performance of the algorithms and data structures. They demonstrate the maximum lookup speed that Concise is able to achieve on a computing device by eliminating the I/O overhead.

2. Click Modular Router [19] is an architecture for building configurable routers. We implement an Concise prototype on Click. It is able to serve as switch that forwards data packets.

3. Intel Data Plane Development Kit (DPDK) [13] is widely used in fast data plane designs. We use a virtualized environment to squeeze both the traffic generator and the forwarding engine on the same physical machine. This prototype is able to serve as a real switch that forwards data packets.

B. Methodology

We compare Concise with three approaches for name switching: (1) Cuckoo hashing [24] (used in CuckooSwitch [42] and ScaleBricks [41]), (2) BUFFALO [36], (3) Orthogonal Bloom filters, and (4) Bloomier Filter. CuckooSwitch [24] is optimized for a specific platform with 16 cores and 40 MBs of cache. ScaleBricks [41] is designed for a

high performance server cluster. We were not able to repeat their experiments on commodity desktop computers. Instead, we compare Concise with (2,4)-Cuckoo hashing, which is their FIB, by reusing the code from the public repository of CuckooSwitch. BUFFALO does not always return correct forwarding actions. The false positive rate is set to at most 0.01%. We also implement a technique called Orthogonal Bloom filters (OBFs) for comparison. It uses a Bloom filter to replace an Othello for classification of two sets X and Y: all names in X hit the Bloom filter. The false positive rate is also set to at most 0.01%. The other design of OBFs is similar to Concise. For Bloomier filter, we use version that use three hash functions to determine the neighborhood of names, and we set the ratio of hash buckets to elements to 1.23. This set of parameter is used for maximum optimization for memory space specified in [7].

We do not include SetSep [11] in this section although it shares some similarity to Othello. The SetSep work [41] does not include an update method and was not proposed for FIBs. Also, there is no explicit update algorithm for SetSep in every work in which it has been used [11][41]. Hence, SetSep cannot be directly used for FIBs and it is not suitable to implement SetSep and compare it with other FIB designs. Actually our experiments using a static version of SetSep show that Concise is faster than SetSep for name lookups.

1) Performance metrics:

Data plane performance metrics are used to characterize the performance of the Concise query structure in switches.

Memory cost: the size of memory needed to store a FIB.

MCQ: the *m*aximum number of *C*ache lines transmitted per *Q*uery. During each memory access, a cacheline (usually 256 bits of data in many architectures) is transmitted from memory to the CPU. It is used to characterize the time cost of a query.

Query throughput: the number of queries that a FIB is able to process per second.

Query throughput under update: the query throughput measured when the FIB is being updated. It reflects the effectiveness of the concurrency control mechanism.

Processing delay: the processing delay of the query structure for a packet. It reflects the ability of the data plane to process burst traffic. Such metric is measured using an event-based simulator on real traffic trace.

Control plane performance metrics characterize the performance of the Concise control structure in the controller.

Construction time: the time to construct a FIB. Note that, for some networks in which G is shared among all switch FIBs such as Ethernet, not every FIB requires the entire construction time. Once G is determined, it can be reused for all switches.

Update throughput: the number of updates that can be processed by the control structure per second. Here, an update may consist in adding a name, deleting a name, or changing the forwarding action of a name.

2) Evaluation environment and settings:

LFSR name generator In the experiments, a series of query packets with different names were generated and fetched by the FIB. One straightforward approach is to feed the FIB with a publicly available traffic trace. However, the time for transmitting the data from the physical memory to the cache is too large compared to the FIB query time. Hence, to conduct more accurate measurement, we use a linear feedback shift register (LFSR) to generate the names. One LFSR generates about 200M names per second on our platform. In addition, we provide event-based simulation using real traffic data to study the processing delay on Concise.

In fact, LFSR gives no favor to Concise because the names are generated in a round-robin scenario, which provides the minimum cache hit ratio. LFSR traffic is actually the *worst* traffic for Concise. On the contrary, in denial-of-service attack traffic, the queries concentrate on one or few names, and they always hit the cache. Hence, the query throughput of Concise in DoS attack traffic may be higher than the value measured with LFSR traffic. We believe the result measured in LFSR traffic reflects the true performance of Concise.

Evaluation Settings In the following section, unless specified otherwise, we evaluate the performance of Concise with 4 parallel query threads. The number of action is set to 256 ($l = 8$). We conduct all experiments on a commodity desktop computer equipped with one Core i7-4770 CPU (4 physical cores @ 3.4 GHz, 8 MB L3 Cache shared by 8 logical cores) and 16 GB memory (Dual channel DDR3 1600MHz).

C. Data plane memory efficiency and MCQ

Table II shows the size of memory of different types of FIBs. For the Cuckoo hash table, we use the (2,4) setting. For BUFFALO, we assume the names are evenly distributed among the actions, which gives an advantage to it. We use the setting $k_{max} = 8$. These settings are all as described or recommended in the original papers [42], [41], [36].

The memory space used by Concise is significantly smaller than that of Cuckoo, BUFFALO, and OBFs. It is only determined by the number of names n and the number of actions, and is independent of the name lengths. Table II also shows the maximum number of cachelines transmitted per query (MCQ) of these FIBs. A smaller MCQ indicates fewer data transferred from the memory to the CPU, which results in better query throughput. Concise always requires exactly two memory accesses per query. The other FIBs may have larger MCQ depending on the name length and number of actions.

D. Memory-mode evaluation

1) Data-plane performance:

Query throughput versus number of names. Figure 5 shows the query throughput of Concise, Cuckoo, BUFFALO, and OBFs. The names are MAC addresses (48-bit).

When n is smaller than 2 million, the throughput of Concise is very high ($>$ 400M queries per second (Mqps)). This is because the memory required by Concise is smaller than the cache size (8M for our machine). When $n \geq$ 2M, the throughput decreases but remains around 100 Mqps. This indicates that if other resources (e.g., I/O and buffer) are not the bottleneck, Concise reaches 100Mqps. The query performance decreases as the size of the query structure exceeds the CPU cache size. We observe similar results when running the evaluation on other machines with different CPUs. Bloomier filter shows lower throughput than Concise. Note

FIB Example			Concise		Cuckoo		BUFFALO		OBFs	
Name Type	# Names	# Actions	Mem	MCQ	Mem	MCQ	Mem	MCQ	Mem	MCQ
MAC (48 bits)	7×10^5	16	1M	2	5.62M	2	2.64M	8	7.36M	15
MAC (48 bits)	5×10^6	256	16M	2	40.15M	2	27.70M	8	112.06M	16
IPv4 (32 bits)	1×10^6	16	1.5M	2	4.27M	2	3.77M	8	10.52M	15
IPv6 (128 bits)	2×10^6	256	4M	2	34.13M	6	11.08M	8	44.82M	16
OpenFlow (356b)	3×10^5	256	1M	2	14.46M	6	1.67M	8	6.72M	16
OpenFlow (356b)	1.4×10^6	65536	8M	2	67.46M	6	18.21M	1024	66.60M	17
File name (varied)	359194	16	512K	2	19.32M	10	1.35M	8	5.47M	15

Table II: Memory and query cost comparison of four FIBs and SetSep. MCQ: maximum # of cachelines transmitted per query.

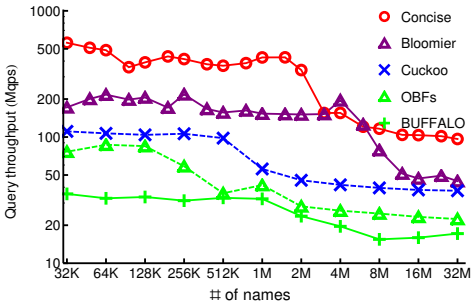


Figure 5: Query throughput versus number of names.

that Bloomier filter is a static solution and the memory space for bloomer filter is smaller than Concise and hence it may show even higher throughput when it is able to fit in the cache. Cuckoo is only about only 20% to 50% of Concise. The results of Cuckoo are consistent with those presented by the original CuckooSwitch paper[1]. Note that the measured time overhead includes that of query generation.[2]

Query throughput versus name length and number of CPU cores. Figure 6 shows the query throughput using different name lengths. Each FIB contains 256K names. As the length grows, the throughput of all types of Concise and Cuckoo FIBs decreases. Note that the memory size of Concise is independent of the name length. Hence, the throughput decrease of Concise is due to the increase of hashing time. One interesting observation is that when the length is a multiple of 64 bits, the query throughput of Concise is slightly increased. This is mainly because the experiments are conducted on a 64-bit CPU. The query throughput grows approximately linearly to the number of used threads, as long as the number of threads does not exceed the number of physical CPU cores of the platform.

Query throughput during updates. Figure 7 shows the throughput of Concise during updates, including name additions, deletions, and action changes. There is only very small decrease of query throughput even when the update frequency is as high as hundreds of thousands of names updated per second. We mark the one-σ (68%) confidence interval of the throughput when there is no concurrent query in Figure 7. Evaluation result shows that the throughput of Concise still remains in its normal range during updates. For Concise with 4M names the throughput downgrade is negligible.

Query throughput versus number of forwarding actions. Figure 8 shows the query throughout of using Concise for one thread for different number of forwarding actions. Using l-POG, Concise is able to represent 2^l forwarding actions. Concise for smaller number of forwarding actions has smaller l value, and hence its memory size is smaller, which leads to a higher query throughput. The throughput is better when the number of forwarding actions equals to 2, 4, 16 or 256. This is because the memory of Othello query structure is better aligned when $l \in \{1, 2, 4, 8\}$.

Processing delay. We conduct event-based simulations of packet processing on the data plane to study the process delay. We simulate a single-thread processor with two-level cache mechanism. The packets are processed in a first-come, first-served fashion. Each packet consists of the header and payload. The packets are put in a queue upon reception and wait to be processed by the prosessor. We measure the processing delay for real traffic data from the CAIDA Anonymized Internet Traces of December 2013 [1]. The average packet rate is about 210K packets per second. In Figure 9, Concise has smaller processing delay than Cuckoo before the 90th percentile, but they have similar tails. To study the processing delay under larger traffic volumes, we replay the trace 100x as fast as the original. Shown as the thin curves, the processing delay of Concise is clearly smaller than that of Cuckoo before the 60th percentile. After that, the two curves are similar, except that Cuckoo has a longer tail. Overall, the processing delay of Concise is very small ($< 1\mu$s) even under high data volumes.

2) Control plane performance:

Construction time. Figure 10 shows the average time to construct the query and control structures for one switch with various number of names. The construction time of Concise grows approximately linearly to the number of addresses. Although the time of Concise is larger than that of Cuckoo and BUFFALO, it is still very small. For 4M names, it takes only 1 second to construct the FIB. Note that the graph G can be reused for all other switches in the network. Hence, network-wide FIB construction only takes few seconds.

Update speed. The update speed indicates the ability to react to network dynamics. All types of network dynamics, including host and link changes, are reflected as name additions, deletions, and action changes in the FIBs. Figure 11 shows the update speed of Concise in number of updates processed per second. We vary the number of names before

[1]The paper [42] showed a throughput 4.2x as high as our Cuckoo results on a high-end machine with two Xeon E5-2680 CPUs (16 cores and 40MB L3 cache). It is approximately 4x as powerful as the one used in our experiments.

[2]In the evaluation of 1M names, each query of Concise takes about 4.5 ns while generating a query takes 4.1 ns.

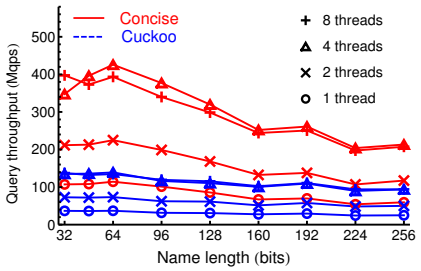


Figure 6: Query throughput versus name length

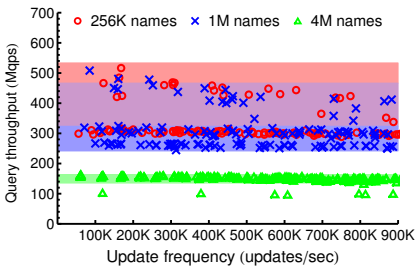


Figure 7: `Concise` query throughput under different update rates

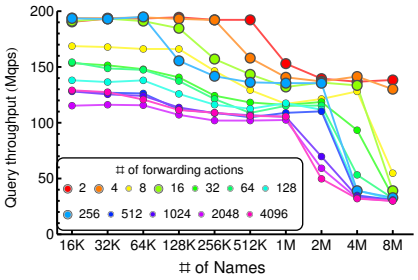


Figure 8: Query throughput versus number of forwarding actions

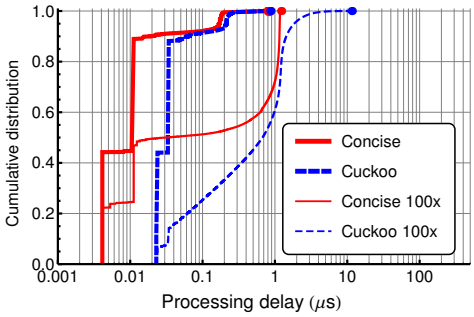


Figure 9: CDF of the processing delay of `Concise` and Cuckoo

update and measure the time used to insert a number of new names. Each run of the experiment is shown as a point in the figure. Note that in a minor fraction of cases the update may cause a reconstruction of Othello, the time overhead of which is comparable to the construction time shown in Figure 10. The amortized speed for `Concise` update is shown in the curve in Figure 11. In most cases, it reaches at least 1M updates per second, which is sufficient for very large networks.

On POG reconstruction. In some rare cases, adding a new name may require reconstruction of the POG when it introduces a new cycle in to the bipartite graph. This may take non-negligible time (0.2 seconds when there are 1M names). Theoretical results show this happens with probability less than $\frac{1.5}{n}$. This value is even smaller in practice (about 1.3 parts per million when there are 1M names). Note that, POG reconstruction may happen only when there is a new name added to the network. *Modifying a forwarding action of a existing name (or removing a name) never results in POG reconstruction.* The line in Fig. 11 shows the average update speed (including the time overhead for reconstruction). POG reconstruction only imposes minor impact on the update speed.

Network-wide shared bipartite graph. For some networks that require every switch to store all destination names such as Ethernet, the name set S is identical for all switches in the network. Hence, all switches in the network may share the same G and $\langle h_a, h_b \rangle$. Constructing and updating the FIBs in all switches only require computing G once. e.g., the phase I of the `construct` procedure (Sec. IV-B1) is only executed *once* for FIBs of all switches in the network. This indicates that the construction time overhead for FIBs of multiple switches can be further reduced. Note that for a single switch, the time used for phase I is about half of the total of `construct`.

	$n=3\times10^5$ 2^8 actions	$n=1.4\times10^6$ 2^{16} actions
Name addition	75.2	107.2
Action change	65.6	88.8

Table III: Entropy of one update message in bits

Communication overhead. We compute the entropy of the information included in update messages in Table III. The update message length grows logarithmically with respect to either the number of names n or the number of actions. The communication overhead of `Concise` is smaller than that of most OpenFlow operations.

Cost of detecting invalid names We also measure the cost of two approaches to detect invalid names. Figure 12 shows that using a 8-bit checksum (marked as Concise+Chk in the figure) has a minor impact on the query performance. We provide more analysis on the approaches in Sec. VII-A.

E. Prototype Implementation and Evaluation

1) Implementation on Click: We implement a `Concise` prototype on Click Modular Router [19]. It receives packets from one inbound port and forwards each packet to one of its 4 outbound ports. Upon receiving a packet, it queries the POG using the address field of the packet, i.e., the name, and decides the outbound port of the packet. In addition, we implement the `(2,4)`-Cuckoo hash table, OBFs, as well as the binary search mechanism on Click. Figure 13 shows the forwarding throughput. The Click modules in each evaluation includes one traffic generator generating packets with valid 64-bit names, one switch that executes queries on the FIB, and packet counters connected to the egress ports of the switch. The experiments are conducted on one CPU core.

Results show that `Concise` always has the highest throughput. When $n < 2$M, `Concise` is smaller than the cache size and the query throughput is about 2x as fast as Cuckoo and 4x as fast as OBFs. When $n \geq 2$M, the throughput of `Concise` is still the highest. Meanwhile, `Concise` uses much less memory, about 10% to 20% of that of Cuckoo, OBFs, and Binary.

2) Implementation with DPDK: We also build a `Concise` prototype on the hardware Environment Abstraction Layer (EAL) provided by DPDK. It maintains a POG query structure. The query structure is initialized during boot up and can

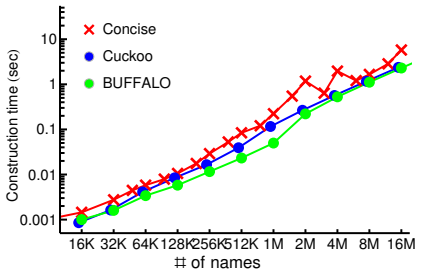


Figure 10: Construction time comparison among three FIBs

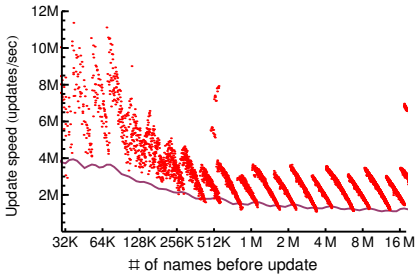


Figure 11: Update speed. Line: avg. spd. including POG reconstruction.

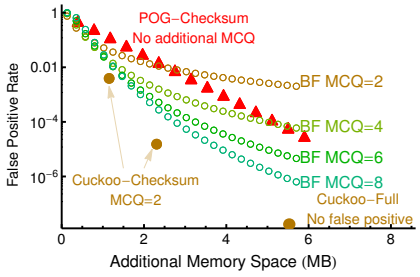


Figure 12: Approaches of detecting invalid names

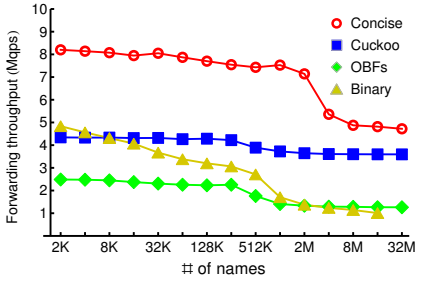


Figure 13: Forwarding throughput comparison on Click

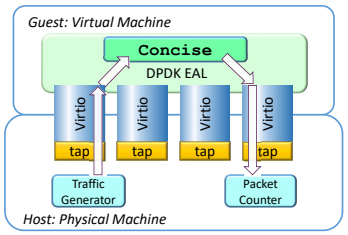


Figure 14: `Concise` prototype on DPDK

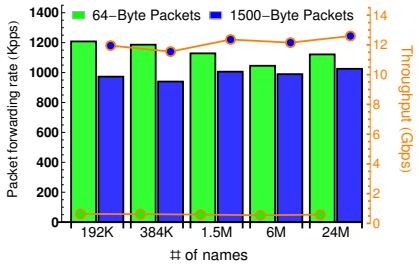


Figure 15: Performance of the `Concise` prototype on DPDK

be updated upon network dynamics. The prototype reads packets from the inbound ports, executes queries on the query structure, and then forwards each packet to the corresponding outbound port.

We implement both the traffic generator and FIB application on the same commodity computer using virtualization techniques. As shown in Figure 14, we create a guest virtual machine (VM) on the host machine using KVM and Qemu to install `Concise`. The VM is equipped with four virtio-based virtual network interface cards. Linux TAP kernel virtual devices are attached to the virtio devices on the host side. The programs running on the host machine communicate with the guest VM via the Linux TAPs. On the host machine, we use a traffic generator program to send raw Ethernet packets to `Concise` running on the VM. The host machine receives the forwarded packets from `Concise` and counts the number of packets using default counters provided by the Linux system.

We measure the throughput of `Concise` with different numbers of names. The barchart in Figure 15 shows that `Concise` is able to generate, forward, and receive more than 1M packets per second, for both 64-Byte and 1500-Byte packets. The forward throughput is at least 12 Gbps for 1500-Byte Ethernet packets. The throughput of Cuckoo is only 60% to 80% of the throughput of `Concise`. The forwarding throughput does not significantly change when the number of names grows or packet length changes. This indicates that the impact of `Concise` on the overall performance is so small that it is negligible compared to the other overheads. The bottleneck of this evaluation is on other parts of processing, e.g., data transmission between the host machine and guest VM. We expect a much higher throughput on physical NICs.

VII. Discussion

A. Properties of Concise for alien names

An alien name is a name that is not in S during `Concise` construction, such name is unexpected for name switching queries. The Othello query of alien names returns a value that is determined by the corresponding value in the memory space of the Othello query structure.

`Concise` is not able to distinguish alien names. This is because in `Concise` the *membership* information about the set of expected names is only maintained by the Othello control structure. The query structure only maintains the *classification* information of the names. In fact, a major part of the memory space for hash table based approaches (e.g, Cuckoo) is used to store copies or digests of the names in order to maintain the membership information. Meanwhile, Bloom filter [7] is a probabilistic data structure widely used to maintain the approximated membership information.

The detailed discussion of the behavior of Othello alien queries is presented in [39]. In the worst case, a packet with unexpected name is forwarded to one of the neighbors of the switch. Compared to the forwarding table miss of Ethernet, which let the packets flood to all interfaces, `Concise` causes no flooding.

In many large-scale flat-name networks the network addresses are all known and there are no unexpected names. However, operators may still choose one or some of the following mechanisms to enable the switches to detect the alien names.

- At an ingress switch, every incoming packet should be checked by a filter or firewall to validate that its destination

does exist in the network. This filter can be implemented as a network function running on the border of the network, and can be integrated with the firewall.
- Maintain a Bloom filter at each of the switches. Packets with valid names pass this filter and are then processed by `Concise` FIB.
- In addition to the l-bit query results, also maintain the checksums for each name in the `Concise` FIBs. Adding checksums will increase the memory size of `Concise`. For r-bit checksums, the overall memory cost of a query structure is $2(l + r)m + O(1)$. Note that as long as $l + r$ does not exceed the word length of the computing platform, the time overhead of all operations remains unchanged.

Assuming there are in total 1M names. Fig 12 compares the memory and computational overheads of the above approaches. The false positive rate can be controlled to be as low as 10^{-5} with $<$ 2MB memory overhead using the filter of Cuckoo with checksums. The performance when using Bloom filters may vary depending on the parameters. We also recommend to utilize the time-to-live (TTL) value of to prevent the packet being forwarded in the network forever.

The unique property of returning an arbitrary value for an alien name may also be useful for `Concise` as a network load balancer: for a server-visiting flow that is new to the network, `Concise` can forward it to one of the servers with adjustable weights.

B. Concise versus Cuckoo and SetSep

`Concise` is essentially a classifier for names, and each class represents a forwarding action. `Concise` does not store the names. Cuckoo stores all names and actions in a key-value store.

SetSep has some properties similar to `Concise`. Both of them do not store names and return meaningless results for unknown names. In ScaleBricks [41], SetSep is only used as a separator to distribute the FIB to different computers, rather than the FIB. Meanwhile, the update scheme for SetSep is not explicitly explained [11], [41], and there is no discussion about handling dynamic FIB size growth.

In addition to the memory size results in Table 1, we show some comparison results of SetSep in what follows. The construction speed of SetSep is slower than that of `Concise` and Cuckoo by more than an order of magnitude: 10 seconds for one single FIB of 1M names in our experiments. We also measure the update speed of SetSep without adding new names, which turns to be less than 10K/s ($<$ 1 % of `Concise`). The query speed of SetSep is higher than that of Cuckoo. SetSep needs to compute $1 + l$ hash values and read $2 + 2l$ values for each query. We implement a static SetSep with 1.4M names and $l = 8$, using 2.19MB memory. Its query throughput is 211 Mqps using 4 threads. In comparison, `Concise` with the same settings uses 4M memory and reaches 470 Mqps.

In addition, we summarize the reasons of the performance gain of `Concise` as follows. (1) Othello does *not* maintain a copy of the names in the query structure. The memory size of the query structure is much smaller than the other solutions. `Concise` demonstrates higher cache-hit rate, which leads to better performance on cache-based systems. (2) The query procedure does not contain any branches (e.g, `if` statements). This helps the CPU to predict and execute the instructions in the query procedure. (3) The efficient concurrency control mechanism further improves the query speed of `Concise`.

C. Example Use Case

`Concise` provides desired FIB properties for many current and future architecture designs that adopt flat names as mentioned in Sec. I. We present a use case where it can be applied in a large enterprise network.

A large enterprise or data center network may include up to millions of end hosts and more VMs [15]. In these networks, internal flows contribute to the most bandwidth, which can be forwarded by `Concise` using destination names on Layer 2. The destination of a packet in this network can only be either a host or a gateway. We require hosts in the network voluntarily check the validity of the packets before sending them out. This can be easily achieved using software firewalls such as *iptables*. As of the gateway, we require it to execute two network functions: (1) For packets going out from the network, perform Layer 3 routing using the external IP of the destination. This is a basic function a router. (2) For packets going into the network, filter out all packets with invalid destinations. This can be implemented by a firewall. The packets will be forwarded using the Layer 2 names of the destinations. In addition, we require all packets in the network to carry a time-to-live (TTL) value to prevent packets from being forwarded forever in case packets with invalid names pass the firewalls.

VIII. Conclusion

`Concise` is a portable FIB design for network name lookups, which is developed based on a new algorithm Othello Hashing. `Concise` minimizes the memory cost of FIBs and moves the construction and update functionalities to the SDN controller. We implement `Concise` using three platforms. According to our analysis and evaluation, `Concise` uses the smallest memory to achieve the fastest query speed among existing FIB solutions for name lookups. As a fundamental network algorithm, we expect that Othello Hashing will be used in a large number of network systems and applications where existing tools such as Bloom Filters and Cuckoo Hashing may not be suitable.

Acknowledgment

The authors would like to thank the comments from Ken Calvert, Xiaozhou Li, as well as the TON reviewers. Ye Yu and Chen Qian were supported by National Science Foundation Grants CNS-1701681 and CNS-1717948. Qin Zhang was supported in part by NSF CCF-1525024 and IIS-1633215.

References

[1] The CAIDA UCSD Anonymized Internet Traces. http://www.caida.org/data/passive/passive_2013_dataset.xml. Date accessed 12/01/2015.

[2] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. of USENIX NSDI*, 2010.

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

[4] H. Asai and Y. Ohara. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. In *Proc. of ACM SIGCOMM*. ACM, 2015.

[5] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *Proc. of ACM SIGCOMM*, 2004.

[6] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with O (1) accesses. In *Proc. of ACM SODA*. Society for Industrial and Applied Mathematics, 2009.

[7] O. B. Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, Ayellet Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables, 2004.

[8] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. ROFL: Routing on Flat Labels. In *Proc. of ACM SIGCOMM*, 2006.

[9] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba. PayLess: A Low Cost Netowrk Monitoring Framework for Software Defined Networks. In *Proc. of IEEE/IFIP NOMS*, 2014.

[10] L. Devroye and P. Morin. Cuckoo hashing: further analysis. *Information Processing Letters*, 86(4):215–219, 2003.

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

[12] B. A. Greenberg et al. VL2: a scalable and flexible data center network. *ACM SIGCOMM CCR*, 09:51–62, 2009.

[13] Intel. Data Plane Development Kit. http://dpdk.org/. Date accessed 01/20/2016.

[14] V. Jacobson, D. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. of ACM CoNEXT*, 2009.

[15] S. Jain and Others. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. of ACM SIGCOMM*, 2013.

[16] S. Janson and M. J. Luczak. Susceptibility in subcritical random graphs. *J. Math. Phys.*, 49(12):125207, 2008.

[17] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. In *Proc. of USENIX NSDI*, 2012.

[18] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *Proc. of SIGCOMM*, 2008.

[19] E. Kohler, R. Morris, and B. Chen. *The Click Modular Router*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[20] X. Liu, Y. Yu, J. Liu, C. F. Elliott, C. Qian, and J. Liu. A novel data structure to support ultra-fast taxonomic classification of metagenomic sequences with k-mer signatures. *Bioinformatics*, 34(1):171–178, 2017.

[21] B. S. Majewski, N. Wormald, G. Havas, and Z. Czech. A Family of Perfect Hashing Methods. *Comput. J.*, jun 1996.

[22] 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 *Proc. of ACM/IEEE ANCS*, 2015.

[23] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol. Technical report, 2008.

[24] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, may 2004.

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

[26] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani. MobilityFirst: A Robust and Trustworthy Mobility Centric Architecture for the Future Internet. *MC2R*, 2012.

[27] J. Saltzer. On the naming and binding of network destinations. RFC 1498, 1993.

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

[29] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy. Scalable Routing on Flat Names. In *Proc. of ACM CoNEXT*, 2010.

[30] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. of ACM SIGCOMM*, 1999.

[31] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. PAST: Scalable Ethernet for Data Centers. In *Proc. of ACM CoNEXT*, 2012.

[32] Y. Wang et al. Wire speed name lookup: a GPU-based approach. *Proc. of USENIX NSDI*, 2013.

[33] T. Yang et al. A shifting bloom Filter Framework for Set Queries. In *Proc. VLDB*, 2016.

[34] D. Yang et al. Difference Bloom Filter: a Probabilistic Structure for Multi-set Membership Query. In *Proc. ICC*, 2017.

[35] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP Lookup Performance with FIB Explosion. In *Proc. of ACM SIGCOMM*, 2014.

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

[37] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proc. of ACM SIGCOMM*, 2010.

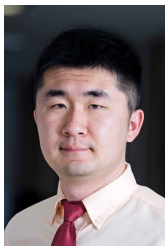
[38] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang. A concise forwarding information base for scalable and fast name lookups. In *Proc. of IEEE ICNP*, 2017.

[39] Y. Yu, X. Li, and C. Qian. Sdlb: A scalable and dynamic software load balancer for fog and mobile edge computing. In *Proc. of the Workshop on Mobile Edge Communications*, MECOMM, 2017.

[40] L. Zhang, *et.al* Named data networking (NDN) project. *NDN Technical Report*, Relatrio Tcnico NDN-0001, Xerox Palo Alto Research Center-PARC, 2010.

[41] D. Zhou, B. Fan, H. Lim, D. G. Anderson, M. Kaminsky, M. Mitzenmacher, R. Wang, and A. Singh. Scaling Up Clustered Network Appliances with ScaleBricks. In *Proc. of ACM SIGCOMM*, 2015.

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



Ye Yu (S'13) is a Ph.D. student at the Department of Computer Science, University of Kentucky. He received the B.Sc. degree from Beihang University. His research interests including data center networks, software defined networking. Especially, he is doing research about applications of fast and memory-effective hashing applications in computer networking systems and data storage.

Djamal Djamal Belazzougui is currently a researcher at CERIST research centre, Algeria. He received an engineering degree from the national high school of Computer science, Algeria, and earned a Phd degree from Paris-VII, Paris-Diderot university, France. He subsequently spent three years as a postdoctoral researcher at the University of Helsinki, Finland. His research topics include hashing, succinct and compressed data structures and string algorithms.

Chen Qian (M'08) is an Assistant Professor at the Department of Computer Engineering, University of California Santa Cruz. He received the B.Sc. degree from Nanjing University in 2006, the M.Phil. degree from the Hong Kong University of Science and Technology in 2008, and the Ph.D. degree from the University of Texas at Austin in 2013, all in Computer Science. His research interests include computer networking, network security, and Internet of Things. He has published more than 60 research papers in highly competitive conferences and journals. He is a member of IEEE and ACM.

Qin Zhang Qin Zhang is currently an Assistant Professor at Indiana University Bloomington. He received the B.S. degree from Fundan University and the Ph.D. degre from Hong Kong University of Science and Technology. He also spent a couple of years as a post-doc at Theory Group, IBM Almaden Research Center, and Center for Massive Data Algorithmics, Aarhus University. He is interested in algorithms for big data, in particular, data stream algorithms, sublinear algorithms, algorithms on distributed data, I/O-efficient algorithms, and data structures.