A Concise Forwarding Information Base for Scalable and Fast Name Lookups

Ye Yu*, Djamal Belazzougui‡, Chen Qian†, and Qin Zhang§
* University of Kentucky; ‡ CERIST; † University of California, Santa Cruz; § Indiana University Bloomington
Email: ye.yu@uky.edu; dbelazzougui@cerist.dz; qian@ucsc.edu; qzhangcs@indiana.edu

***Abstract*—Forwarding information base (FIB) scalability and its lookup speed are fundamental problems of numerous network technologies that uses location-independent network names. 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.**

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 [23], 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 [23] is one of the early proposals that suggest the separation of network identifiers and locators. We summarize an (incomplete) list of reasons of 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[15], [12], [27], 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 [19], Layered Naming Architecture [5], and Mobility-First [22].
- AIP [3] and XIA [20] apply 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 [32].

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 [30], [10], [31], 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 upgrade when the network grows. Hence, the cost of network construction and maintenance can be significantly reduced. For 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

978-1-5090-6501-1/17/$31.00

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 [33]	$O(n)$	$\sim 1.1n(L+\log w)$	$O(1)$	Up to 8 memory reads per query.
SetSep [32]	$O(n \log w)$	$(2+1.5 \log w)n$	$O(\log w)$	No method for updates. Not designed as FIB in [32].
BUFFALO [30]	$O(nt)$	αn	$O(tw)$	Probabilistic results. False positive ratio affected by t and α.
TSS [26]	$O(n(t+\log w))$	$O(n(t+\log w))$	$O(t)$	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**.

Bloom filters [30], [18].

`Concise` is built on a new network algorithm named Othello Hashing. Othello applies the theoretical studies on minimal perfect hashing [17], [6]. Different from the theory studies, Othello is carefully designed for network applications and supports network dynamics. 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 minimum perfect hashing schemes to network applications with update functionalities. `Concise` is designed for name switching, so it does *not* support IP prefix matching.

`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 [16], and Intel Data Plane Development 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 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.

Related Work

Location-independent network names. Separating network location from identity has been proposed and kept repeating for over two decades. Numerous network architectures suggest this concept. As discussed in Sec. I, a number of new network architectures adopts location-independent names. A location-independent name can be a MAC address, a tuple consisting of several packet header fields, a file name, a TEID [32], etc. To route packets for flat names, ROFL [9] and Disco [25] propose to use compact routing to achieve scalability and low routing stretch. `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 [33] use carefully revised Cuckoo hash tables [21] to reach desirable performance on specific high-end hardware platforms. ScaleBricks [32] 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 [30], [18]. However, they may forward packets incorrectly due to the false positives in Bloom filters, causing forwarding loops and bandwidth waste. For IP lookups, SAIL [29] 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) [26] is widely used for name matching with multiple files, such as in OpenVswitch and PIECES [24]. It is not designed for flat-name switching. Other solutions uses hardware to accelerate name switching. For example, Wang *et al.* [28] uses GPU to accelerate name lookup in Named Data Networks.

Minimal perfect hashing. The data structure used in this work, Othello, is inspired by the studies on minimal perfect hashing. In particular, MWHC [17] is able to generate order-preserving minimal perfect hash functions using a random graph. MWHC is also presented as Bloomier Filter in [7]. The differences between Othello and these studies include: (1) Both MWHC and Bloomier Filter are designed for static scenarios and they do not support frequent updates like Othello does. (2) Othello uses a bipartite graph instead of a general graph. This design allows much simpler concurrency control mechanism. (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 [17] and Bloomier Filter is designed for approximate evaluation queries [7].

Design Overview

Consider a network of n hosts identified by unique names. The hosts are connected by SDN-enabled switches. 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

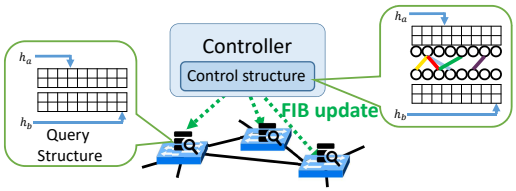


Figure 1: Network Overview of Concise

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 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 use minimal perfect hashing 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.

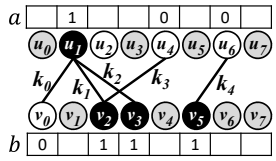
OTHELLO HASHING

In this section, we describe the Othello data structure. Inspired by the MWHC minimal perfect hashing algorithm [17], 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.

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.



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.

- 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 $\boldsymbol{b}$.

Fig. 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 of n. However Option 2 provides slightly better empirical results.

Othello supports the query operation. 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 $\boldsymbol{b}$ is 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 $k \in Y$, $\boldsymbol{a}[h_a(k)] \neq \boldsymbol{b}[h_b(k)]$.

Othello Operations

Othello is maintained via the following operations.

- 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.

Construction

The construct operation takes the input of 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 a 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 be 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 associated with no edges, we do not care about the value of the corresponding bit because it does 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: Deciding 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 also consider 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 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.

Using the techniques described in [8], we get the following conclusion. Given a set of names $S = X \cup Y$, $n = |S|$ and assuming that 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 will be acyclic with probability $\sqrt{1 - c^2}$**, where $c = \frac{n}{\sqrt{m_a m_b}} < 1$.

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 $O(n)$.

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. 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 colors.

Complexity Analysis.

We show the time complexity of `add` is $O(1)$. From the results in Theorem 3.3(i) in [14], we get the following conclusions.

- In an Othello bipartite graph $G = (U, V, E)$, randomly select a node $w \in U \cup V$. Let $\texttt{cc}(w)$ be the connected component containing w. When $c = \frac{n}{\sqrt{m_a m_b}} \leq 0.75$, $\mathrm{E}[|\texttt{cc}(w)|] \leq 4$ with high probability as $n \to \infty$.
- In the procedure of adding one name into an Othello with n existing names, the execution falls in Case I with probability at most $\frac{1.5}{n}$ when $n \to \infty$.

  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 $n \leq 0.65 m_b$.

`add` fall in Case I only in a very rare circumstance. In this case, the `construct` algorithm is executed in $O(n)$ time. In Case II, the values correspond to vertices in one component is updated in $O(E[|\texttt{cc}|])$ time. Hence, the expected time complexity is $\frac{1.5}{n} \cdot O(n) + O(E[|\texttt{cc}|]) = O(1)$.

Othello size growth. After adding a name into Othello, $n = |S|$ grows and may violate $m_a \geq 1.33n$ and $m_b \geq n$. 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 $E[|\texttt{cc}|]$ 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 $m_b \geq n$.

Set change for a name

Operation $\texttt{alter}(k)$ means to move k from X to 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 $\texttt{alter}(k)$. We only need to change the coloring plan of the connected component including 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 $O(1)$.

Name deletion

$\texttt{delete}(k)$ can be done by simply removing the edge $(u_{h_a(k)}, v_{h_b(k)})$ in the graph G. The bitmaps $\boldsymbol{a}$ and $\boldsymbol{b}$ are not modified because the values of $\tau(k)$ after deleting k do not matter any more. The time complexity is $O(1)$.

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 element is 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 its local memory, namely the *Query structure*. Storing this six-tuple takes $2m + O(1)$ bits memory space. The time cost for each query of Othello is equal to the sum of the cost of computing two hash values, twice 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 executes query on the query structures.

Summary of Othello Properties

An Othello is decomposed to 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 hashes 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 to 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 to $d > 2$ disjoint sets, while still requires small memory and constant query time.

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, which is skipped due to page limit.

Extension of Othello for Name Switching

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 $S = Z_0 \cup Z_1 \cup \cdots \cup Z_{2^l-1}$. For a name $k \in S$, query of the l-POG 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 $(\tau(k) \mod 2^i) < 2^{i-1}$, this indicates that 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 is essentially operations on 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 is 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 $O(n)$.

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 [33], 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 stronger 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.

Implementation and Evaluation

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

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[16] 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 a same physical machine. This prototype is able to serve as a real switch that forwards data packets.

Methodology

We compare Concise with three approaches for name switching: (1) Cuckoo hashing [21] (used in CuckooSwitch [33] and ScaleBricks [32]), (2) BUFFALO [30], and (3) Orthogonal Bloom filters. CuckooSwitch [21] is optimized for a specific platform with 16 cores and 40 MB cache. ScaleBricks [32] is designed for a high performance server cluster. We are 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 new 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.

We do not include SetSep [11] in this section although it shares some similarity to Othello. The SetSep work [32] includes no update method and was not proposed for FIBs. There is no explicit update algorithm for SetSep in every work it has been used [11][32]. 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 in name lookups.

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 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 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 can be processed by the control structure per second. Here, an update may be adding a name, deleting a name, or changing the forwarding action of a name.

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 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).

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 [33], [32], [30].

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

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
MAC (48 bits)	3×10^7	256	96M	2	321.23M	2	166.23M	8	672.34M	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 number of cachelines transmitted per query.

accesses per query. The other FIBs may have larger MCQ depending on the name length and number of actions.

Memory-mode evaluation

Data-plane performance

Query throughput versus number of names. Fig. 3 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)). It 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 still 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. Cuckoo has the highest throughput among the remaining three FIBs but its 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]

Cost of detecting invalid names We also measure the cost of two approaches to detect invalid names. 3 shows that using a 8-bit checksum (marked as Concise+Chk in the figure) has a minor impact on the query performance. We provide some more analysis on the approaches in Sec. VII-A.

Query throughput versus name length and number of CPU cores. Fig. 4 shows the query throughput using different lengths of names. Each FIB contains 256K names. The 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 raised. This is mainly because the experiments are conducted on a 64-bit CPU. The query throughput grows approximiately in proportional 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.

[1]The paper [33] 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.

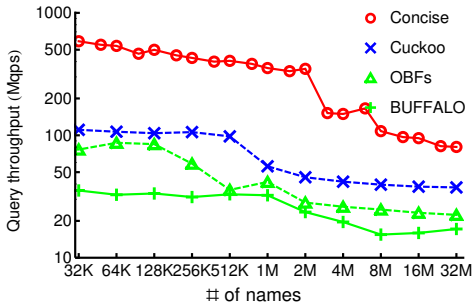


Figure 3: Query throughput versus number of names.

Query throughput during updates. Fig. 5 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 Fig. 5. 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.

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 Fig. 6, 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.

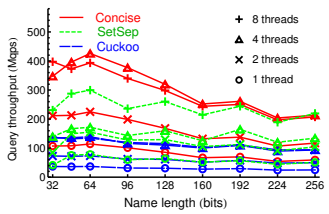


Figure 4: Query throughput vs. name length

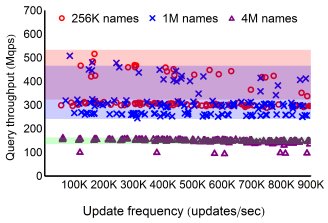


Figure 5: `Concise` query throughput under different update rates

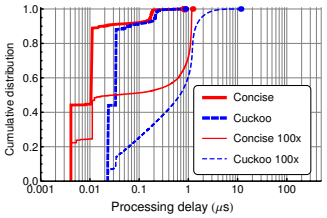


Figure 6: CDF of the processing delay of `Concise` and Cuckoo

Control plane performance

Construction time. Fig. 7 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 linear 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 a few seconds.

Update speed. The update speed of 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. Fig. 8 shows the update speed of `Concise` in the number of updates processed per second. We vary the number of names before 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. 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. 8 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. Note that for a single switch, the time used for phase I is about half of the total of `construct`. This property reduces the construction time overhead for FIBs of multiple switches.

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.

	$n=3\times10^5$, 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

Prototype Implementation and Evaluation

Implementation on Click

We implement a `Concise` prototype on Click. It receives packets from one inbound port and forwards each packet to one of its several 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. Fig. 9 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.

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 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 a same commodity computer using virtualization techniques. As shown in Fig. 10, 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

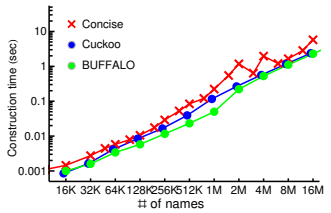


Figure 7: Construction time comparison among three FIBs

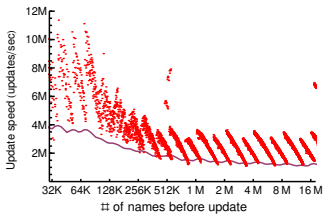


Figure 8: Update speed. Line: avg. spd. including POG reconstruction.

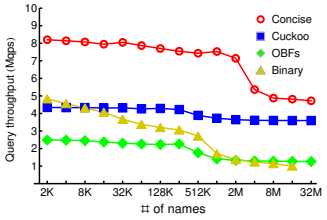


Figure 9: Forwarding throughput comparison on Click

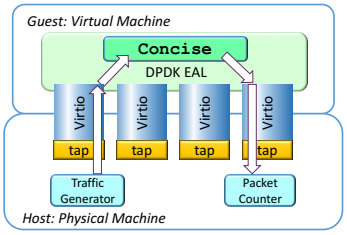


Figure 10: `Concise` prototype on DPDK

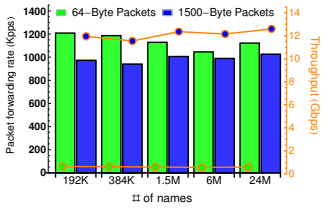


Figure 11: Performance of the `Concise` prototype on DPDK

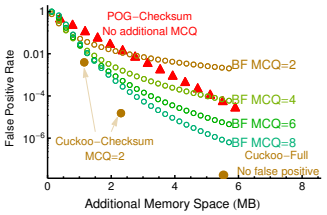


Figure 12: Approaches of detecting invalid names

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 Fig. 11 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% compared to `Concise`. The forwarding throughput has no significant changes when the number of names grows or packet length changes. This indicates the impact of `Concise` to 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.

Discussion

Deal with Alien Names

An alien name is a name that is not S during `Concise` construction. Querying an alien name may result in an arbitrary forwarding action. Compared to the forwarding table miss of Ethernet, which let the packets flood to all interfaces, `Concise` causes no flooding. Operators may choose one or some of the following mechanisms 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 is 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 remain unchanged.

Assuming there are in total 1M names. Fig 12 compares the memory and computational overhead of the above approaches. The false positive rate can be controlled as low as 10^{-5} with $<$ 2MB memory overhead using the filter of Cuckoo with checksums. The performance of 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.

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 store no names and return meaningless results for unknown names. In ScaleBricks [32], 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], [32], 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 as follow. 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. As comparison, `Concise` with 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`.

CONCLUSION

`Concise` is a portable FIB design for name switching, 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.

ACKNOWLEDGEMENT

The authors would like to thank the comments from Ken Calvert, Xiaozhou Li, as well as the anonymous ICNP 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.

[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] F. C. Botelho, N. Wormald, and N. Ziviani. Cores of random r-partite hypergraphs. *Inf. Process. Lett.*, 112(8-9):314–319, apr 2012.

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

[10] 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.

[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/.

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

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

[16] E. Kohler, R. Morris, and B. Chen. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, 2000.

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

[18] 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.

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

[20] D. Naylor and Others. XIA: Architecting a More Trustworthy and Evolvable Internet. *SIGCOMM CCR*, 44(3):50–57, 2014.

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

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

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

[24] 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.

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

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

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

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

[29] 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.

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

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

[32] 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.

[33] 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.