Fast and Efficient Lookups via Data-Driven FIB Designs

Sachin Ashok*
University of Illinois at Urbana-Champaign
Champaign, IL, USA
sachina3@illinois.edu

Aditi Partap*†
Stanford University
Stanford, CA, USA
aditi712@stanford.edu

Ammar Tahir*
University of Illinois at Urbana-Champaign
Champaign, IL, USA
ammart2@illinois.edu

ABSTRACT

With the rapidly growing number of hosts connected to the internet, there is an ever-increasing demand for fast and inexpensive switch memory. At the same time, the number of network functions handled at the switch, especially in the case of a programmable switch, is increasing steadily (e.g., for the purposes of routing, telemetry, load balancing), which require dedicated memory. Various compact and efficient data structures (e.g., Bloom filters [15], ludo hashes [10], cuckoo filters [3]) have been proposed in the past to address these needs. However, these data structures can provide varying performance depending on the distribution of the actual key-value pairs they store. In addition, several of these data structures are probabilistic in nature and hence also trade-off on accuracy to achieve a lower memory usage.

In our work, we propose using data-driven approaches to analyze these key-value pairs (i.e., FIB lookup data) for patterns which can aid in building more informed FIB designs. Primarily, we argue that using an ensemble model comprising of hash tables and Bloom filters (the composition as dictated by the data) can better meet the specific requirements (processing speed, available memory, accuracy level) of the given switch. In this paper, we present a spectrum of designs that are possible within this space and implement one specific prototype. Finally, we present preliminary evaluation of this prototype using enterprise network data to support our proposal.

CCS CONCEPTS

• **Networks** → **In-network processing**; **Routers**; **Bridges and switches**; *Naming and addressing*.

KEYWORDS

forwarding information base; bloom filters; hash tables; decision trees

ACM Reference Format:
Sachin Ashok, Aditi Partap, and Ammar Tahir. 2022. Fast and Efficient Lookups via Data-Driven FIB Designs. In *ACM SIGCOMM 2022 Workshop on Future of Internet Routing & Addressing (FIRA '22), August 22, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3527974.3545728

*Equal contribution.
†Work done while at UIUC.

FIRA '22, August 22, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9328-7/22/08…$15.00
https://doi.org/10.1145/3527974.3545728

1 INTRODUCTION

Packet forwarding in typical network switches consists of four steps (in the simplest form): receive the packet, extract the forwarding information (e.g., destination IP address) from the packet header, find the next hop for the packet by querying a lookup table, and forward the packet to an output port corresponding to that next hop. Switches make use of the Forwarding Information Base (FIB), which maps destination addresses to next hops/output ports, to make such forwarding decisions. To support ever increasing network speeds, the FIB needs to be stored using a data structure which is efficient to lookup, and scales well with increasing number of hosts. Such a data structure needs to have a small memory footprint. Upgrading switches with faster and more memory is not a viable solution as it is cost prohibitive.

There are various approaches used today to implement FIBs which trade-off between memory and processing speed. Hashing-based solutions like Ludo hashing [10] are fast as they allow for $O(1)$ accesses to make the forwarding decision, however they require large memory to over-provision for collisions. On the other hand, tree-like data structures are useful to do longest prefix matching (LPM) and are much more space-efficient since they do not need to overprovision memory like hashing based solutions. However, the lookups require tree-traversal, i.e., multiple memory accesses which make such solutions slow. Neither of these designs scale well with the rapidly growing number of hosts on the Internet and the increasing link capacities. The growing number of hosts require larger hash tables, whereas the increasing link speeds mean that the switches run the risk of becoming a bottleneck if their processing speed cannot scale because of slow lookups. Finally, upgrading switches with larger and faster memory is cost-prohibitive, so there still remains a constant struggle to find FIB designs which can satisfy both the memory as well as the processing speed requirements.

As a response to this struggle, probabilistic data structures (e.g., Bloom filters, cuckoo filters) have been proposed in the past which achieve both speed as well as memory-efficiency by trading off accuracy. BUFFALO[15] proposed a forwarding architecture where a Bloom filter is provisioned for each output port and each individual Bloom filter is responsible for answering whether a packet should be forwarded onto its corresponding output port. Upon receiving a packet, BUFFALO performs parallel lookup in all of these Bloom filters for the packet's destination IP address to find a match. Subsequently, the packet is forwarded to a port that has a match. This approach results in false positives, however picking a port randomly between multiple matched ports each time guarantees

that packet reaches its destination with constant bounded stretch. One problem BUFFALO does not address is the feasibility of performing too many parallel Bloom filter lookups. It is very common for switches to have upto 64 output ports. Therefore, it becomes difficult to perform so many lookups in parallel for each packet owing to limited processing capacity of commodity switches. This makes it difficult to maintain a high processing rate.

To address these concerns about BUFFALO, we take a data-driven approach to think about FIB design. Initially, we wanted to find learnable patterns in forwarding decisions so that we could summarize forwarding in the form of compact machine learning models. Such a FIB design would be space efficient as well as fast, and it would not have same scalability concerns as Buffalo. We analyzed real-world FIB data from a large organization's network. While we did not find significant enough patterns that could be learned through machine learning techniques, we still observed some other patterns that can be leveraged to improve FIB data structure. For example, we observed that not all the output ports on a switch are equally popular. In many cases a few ports have a majority of destination IP addresses mapped to them (which we call *heavy hitter ports*). Based on this observation, we propose a new FIB mechanism that tries to achieve a sweet spot between memory consumption and processing speed. Our design also uses Bloom filters to make forwarding decisions similar to BUFFALO, but we use them only for the heavy hitter ports, whereas for all the other ports we maintain a small hash table thereby limiting the memory usage. To determine which IP addresses may be mapped to heavy hitter ports, we explored a variety of different options, e.g., using a simple machine learning based classifier, Bloom filters, and learned Bloom filters [12]. Finally, while our current approach chooses to focus on improving BUFFALO's design by using data-driven insights, the general idea to inform the construction of such hybrid FIB designs via data extends beyond it.

We implemented our FIB design in C++ and evaluated against simulated workloads based on an enterprise's FIB data. We compare it with the available open-source implementation of Ludo [1]. We found it to be $1.6 - 8\times$ more space efficient than a space-optimized hashing techniques like Ludo with $1.5 - 2\times$ degradation in query time compared to Ludo.

2 APPROACH

We take a data-driven approach to FIB design. We first summarize results from our analysis of the FIB data in 2.1 and then present our data-driven design in 2.2.

2.1 Data Analysis

First, we analyzed network data from an organization network, which comprises of forwarding information from 10s of core routers and 100s of edge routers. We focused on the FIBs of the core routers, since they have much higher number of entries.

Lack of Patterns: Our initial goal was to learn patterns in forwarding data through machine learning techniques. This is helpful because an ML based FIB would be space efficient as well as fast. However, even after using different machine learning techniques, we could not faithfully model FIB forwarding decisions (see table 1). The best accuracy we could get was 26.5%. Learning forwarding decisions is difficult because of scarcity of learnable patterns in FIB data.

ML Model	Average Accuracy
Decision Trees	26.5%
Random Forest	26.47%
Decision Trees + AdaBoost	19.13%
Support Vector Machines	25.13%
Linear Regression	25.19%

Table 1: Accuracy of different machine learning techniques in modeling FIB's forwarding decisions.

Heavy Hitter Ports: Digging deeper into the data, we noticed a disparity in how IP prefixes are mapped to output ports – not all output ports are equally popular. In fact, we observed a very common pattern: a handful of output ports (less than 10) have most IP prefixes mapped to them, we call such ports *heavy hitter ports*. Whereas majority of switches only have handful of IP prefixes assigned to them. Figure 1 shows such mapping for a representative switch from our data. This pattern makes sense because the heavy hitter ports are typically the network's outfacing ports which take traffic from the devices in the network to the Internet. This pattern thus may not just be specific to the network of our dataset, but a common theme to most enterprise networks. We also noticed that it is a little easier to model classification of IP prefixes to heavy hitter or non heavy hitter ports (Table 2). First, it is an easier problem to solve because we just have binary classification per IP prefix. Secondly, patterns can emerge from distinction between internal IP addresses and external IP addresses. We get an accuracy of 93.39% with decision trees along with AdaBoost. This is the model we reuse for the rest of the paper unless stated otherwise.

2.2 Design

We want to have a FIB design that is both space efficient and fast. We take inspiration from Ludo and BUFFALO. Hashing based FIB used in Ludo is very fast, but takes up a lot of space. On the other hand, the Bloom filter based design of BUFFALO is quite space efficient but does not perform well with larger number of output ports.

2.2.1 A Hybrid Design. Since we know that most of the IP prefixes map to a small number of heavy hitter ports, we can use one Bloom filter per heavy hitter port. It is practical to perform parallel lookups in these many Bloom filters. And since rest of the ports do not have a lot of IP prefixes mapped to them, and hence, are not the memory bottleneck - so they can be stored in a small hash table. We can control the number of Bloom filters in our data structure by varying a parameter called *heavy hitter threshold* (θ), it is a real number between 0 and 1, that determines when a port is determined to be a heavy hitter. If the number of entries mapping to a port are greater than $\theta \times S$, where S is the total number of entries in the FIB, then this port is determined to a heavy hitter port.

[1]We did not find an open source implementation for Buffalo, but in our evaluation we show our design's comparison with a proxy design for a Buffalo based solution.

2.2.2 Heavy Hitter Classifier. At this point, any given destination IP address could be matched to either heavy hitter ports (Bloom filters) or non heavy hitter ports (hash table). Looking up in parallel for all packets in both of them may not be ideal as it will significantly slow down the processing capability of the router. So, we need a light weight classifier at the top that helps decide which data structure to search in, for each packet. We discuss a few possible designs for this below:

ML Models: As discussed, there are some patterns in classifying IP prefixes as mapping to heavy hitter ports. We were able to build decision tree models which had accuracy of upto 92-93.39%. This classifier can however result in false positives and false negatives. In the case of wrong classification, we might have to perform sequential lookup in both data structures - i.e. hash table and Bloom filters. A 92% accuracy means that 8% queries would require such sequential lookup. The plus point of using ML however is that in terms of memory it scales well with respect to size of FIB data.

Bloom Filter: Another option is to use a Bloom filter to store the set of all IP addresses that map to heavy hitter ports. Bloom filters can also have false positives, but we can increase size to reduce false positive rate [2]. This however means that as size of FIB data increases, we will need to increase the size of Bloom filter accordingly. The false positive in classifier bloom filter can be corrected if heavy hitter bloom filters do not produce a match, in which case we can just do a sequential lookup in hashtable.

Learned Bloom Filter: Learned Bloom filter[12] gets best of both worlds. Essentially, it consists of a ML model at top and all the *negative* [3] decisions made by ML model are checked by a backup Bloom filter which only stores the model's false negatives. The size of the backup Bloom filter would be really small; if model has a false negative rate of 5%, the size of the backup Bloom filter would also be 5% of the overall size. This solution thus has no false negatives, and in terms of memory, learned Bloom filter scales quite well. Similar to BF and ML classifiers, we can address false positives in the classifier by doing a sequential lookup in hashtable.

2.2.3 Optimizing Memory Usage. The classifier for heavy hitters needs to be accurate enough, since a misclassification can lead to serial lookups in both the Bloom filters as well as the hash table. Secondly, the false positive rates of the bloom filters for the HH ports should also be low, since their errors can lead to faulty forwarding decisions. Hence, we formulate an optimization problem to tune the sizes of the HH bloom filters (and the bloom filter used in HH classification) so as to minimize their false positive rates given limited switch memory. We solve for the optimal false positive rates for all the bloom filters using KKT conditions, similar to BUFFALO's approach. We refer to N as the number of heavy hitter ports, then, we have N Bloom filters, with false positive rates $f_1, ...f_N$ and with memory usage $m_1, ...m_N$. For the case where the HH classifier uses a bloom filter, we denote its false positive rate by f_h, and the total memory used by the classifier as m_h. Let M denote the total memory available in the switch/router, and w_h be the relative weight (or importance) of the accuracy of the HH classifier as compared to the accuracy of the Bloom filters for all HH ports. Then, our objective function is given by :

$$\min\ w_h * f_h + (1 - \prod_{i=1}^{N}(1 - f_i)) \tag{1}$$

Here, the second term represents the overall false positive rate of all N Bloom filters combined. For small values of f_i, the above expression can be approximated as follows:

$$\min\ w_h * f_h + \sum_{i=1}^{N} f_i \tag{2}$$

We add a constraint for total memory usage:

$$\sum_{i=1}^{N} m_i + m_h \leq M \tag{3}$$

We also add the relationship between each bloom filter's memory used and false positive rate, as an equals constraint. More details on our formulation, and how we solve the optimization problem, are available at [1].

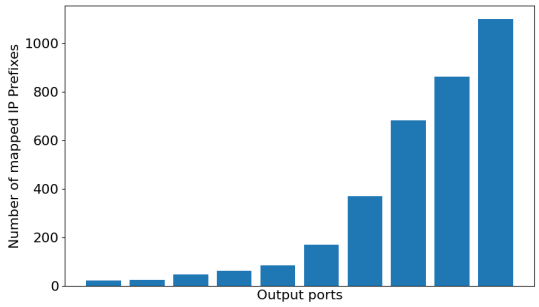

Figure 1: A minority of ports have most IP prefixes assigned to them. This plot shows 10 most popular ports in one of the switches in our dataset.

ML Model	Average Accuracy
Decision Trees	86.98%
Random Forest	87.93%
Decision Trees + AdaBoost	93.39%
Support Vector Machines	89.69%
Linear Regression	83.53%

Table 2: Accuracy of different machine learning techniques to learn whether a given IP prefix maps to a heavy hitter port or not.

3 EVALUATION

We evaluate our proposed design and compare against alternatives mentioned previously.

Dataset. Our dataset comprises of FIB snapshots spanning 5 years of data collected from 10s of core and 100s of exit routers from an enterprise network. For each router, the FIB consists of 100s of entries mapping destination IP prefixes to outgoing ports.

[2]Bloom filters will have false positives but not false negatives.
[3]By negative we mean, for any input model deciding that *input* is not a heavy hitter.

Router #	Workload	Memory (bytes)		Accuracy		Query time (ns)	
		Ludo	Our design	Ludo	Our design	Ludo	Our design
1	Uniform	27500	**16750**	1	0.995	**620**	1020
1	Zipfian	27500	**16750**	1	0.995	**625**	980
2	Uniform	23200	**13800**	1	1	**620**	1060
2	Zipfian	23200	**13800**	1	1	**610**	950
3	Uniform	15800	**2200**	1	1	**610**	1210
3	Zipfian	15800	**2200**	1	1	**610**	1210

Table 3: Results for our FIB design compared against Ludo Hash across several representative core routers and workload types.

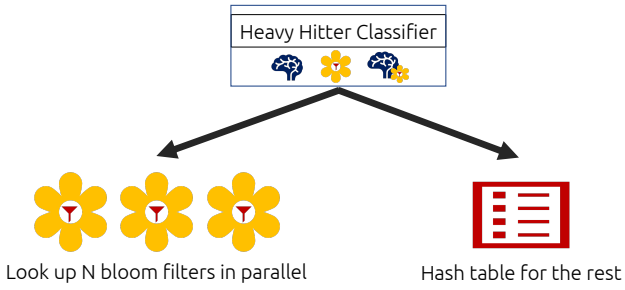

Figure 2: Our FIB design which uses a heavy-hitter classifier to direct next hop lookups towards either the bloom filter or the hash table component at which stage the lookup is performed in-parallel.

Implementation. We implemented our design in C++, wherein we use different types of HH classifiers, and re-use Ludo's Hash tables and Bloom filters implementation to store non-HH and HH entries respectively. We take as input the memory available on the router, and tune the false positive rates of all the bloom filters accordingly, using the solution to our optimization problem. We use $w_h = 1/N$, and $\theta = 0.01$ (i.e. a port is a HH if more than $0.01 * S$ IP addresses map to it). We ran all our experiments on a t2.micro EC2 VM on AWS.

Methodology. For performance testing, we query each FIB design with keys (i.e., with IPs). To enable the testing, we first create and populate the FIB data structures for our design and baselines with a specific router's FIB entries.
The query workloads are of two types:

(1) Uniform: All keys are requested with equal probability.
(2) Zipfian: Keys are requested with biased probabilities.

Finally, we analyse performance in terms of the memory usage, accuracy of the lookup, and the query time.

For our design, we analyze performance of machine learning (ML), Bloom filter (BF) and learned Bloom filter (ML + BF) as HH classifiers respectively.

Results. Figure 3 highlights the performance of several designs populated with core router #1's FIB data. We evaluate our design with different classifiers for heavy hitters. For each classifier, we also vary the value of memory available. As the amount of available memory increases, we can use bigger Bloom filters for all HH ports as well as for HH classifier (i.e. larger memory footprint) which give us lower false positive rates and hence improves our accuracy. Also, with less errors, there are fewer IP addresses for which we need to do serial lookup in both data structures, hence improving performance as well. Our results confirm these trends.

Comparison to Ludo Hashing. Ludo Hashing naturally achieves the lowest query time due to its $O(1)$ accesses to its hash table and its accuracy is 1 since it does not use probabilistic data structures. However, as mentioned previously, Ludo Hashing requires considerable over-provisioning and hence is undesirable when router memory is at a premium. Considering the points at 18000 bytes, Our design (with BF or ML+BF classifier) achieves the same accuracy as Ludo (i.e. 1) in 1.5x less memory, with 1.6-2x degradation in mean query time.

Comparing different HH Classifiers. For our design, using Bloom filter as the HH classifier performs the best in terms of accuracy, memory and query time. However, on a different dataset the ML+BF classifier may outperform it especially since it theoretically scales better than simple Bloom filters. Table 3 summarizes results across a few different core routers and we see similar trends.

4 DISCUSSION

Consequence of inaccuracy. In our ensemble FIB design, inaccuracy in the form of a false positive can arise from any Bloom filter component. For the Bloom filters used for HH classification, an error could lead to serial lookup in both data structures, which we saw in the results. But, the consequence of a false positive in the Bloom filters for HH ports (i.e., incorrectly selecting a next hop among multiple next hops) can lead the packet astray. However as per BUFFALO [15], by randomly selecting the next hop from all the matching next hops, excluding the interface on which the packet arrived, guarantees that packet reaches its destination with constant bounded stretch.

Key addition. The bloom filters for HH ports support updates by design - a new key can be added in constant time. For the Ludo Hash component of our ensemble FIB design, we can leverage its capability to dynamically add keys during run-time. When a new key needs to be added, we can simply add it to the Ludo Hash or to the Bloom Filter of the corresponding port.

FIB design updation. The FIB data can be periodically analyzed to re-classify whether each output port is a heavy-hitter or not. Also, both the bloom filters and Ludo hash might need to be re-sized if the number of keys increases. Subsequently, a possibly different

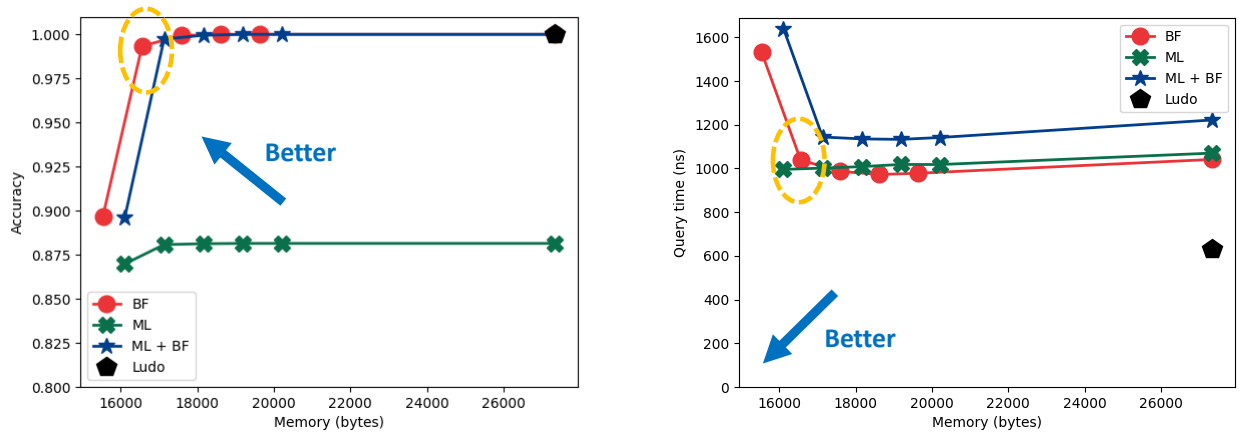

Figure 3: Evaluation on a representative core router. The left plot shows memory-accuracy tradeoff of our design against Ludo. The right plot compares query time. Our design has one curve for each classifier (ML for Machine Learning, BF for Bloom Filter and ML+BF for Learned Bloom Filter classifier) with 5 data points each for different values of Bloom filter sizes (and therefore assumed available memory).

number of Bloom filters (with different sizes) and a differently sized hash table can be initialized and populated with the new and updated FIB data. We assume capabilities of a programmable switch (e.g., Barefoot Tofino) to achieve this. If we use an ML model for HH classification, we might also need to re-train it - this could potentially be done at the edge and the models could be pushed to the switches. We leave this to future work.

Resource driven optimization When deciding on number of Bloom filters (N), we can also take into account available resources. If the switch has ability to run only m number of parallel threads then N should not be greater than m to ensure line rate. Similarly, if the switch is memory bound, we can push more load towards Bloom filter at cost of higher query time and drop in accuracy.

5 RELATED WORK

Traditionally forwarding information base has been designed using tries or trees [4, 6, 11, 14]. Tries allow longest prefix matching by traversing the tree. However, tree traversal can be slow and therefore such solutions suffer with processing rates.

To improve the lookup time, many solutions rely on hashtables and key-value stores. Hashing schemes like Cuckoo hashing [8, 16] can allow $O(1)$ lookups. However, hashtables require memory overprovisioning to avoid or resolve hashing conflicts. Most hashing based solutions thus require storing keys along with the values in the table to resolve conflicts. This further adds to memory requirements of such solutions. There are solutions to address this problem in form of Ludo Hashing [10] and other similar solutions [3, 13]. Such solutions perform different compression techniques on keys to reduce overhead of storing them. Despite of this, hashing based solutions still require memory overprovisioning to reduce collisions to ensure $O(1)$ lookups.

Bloom filters have also been used to make space efficient yet fast FIBs [2, 7, 15]. Buffalo e.g. maintains a Bloom filter for each outgoing port and stores in it the IP addresses mapped to that port. Lookups for IP addresses then require parallel lookups in all these Bloom filters. Bloom filter based solutions have also been extended to provide ability to do longest prefix matching [2, 5, 7].

Hardware based solutions like TCAM give the ability to do longest prefix matching in $O(1)$ through specialized hardware without overprovisioning like hashtables[9]. However, such hardware is expensive and difficult to program.

6 CONCLUSION

This paper presents a case for using data-driven approaches to better inform the design of a router's forwarding information base. We present our design and evaluate our prototype using network data from an enterprise network comprising of FIB data from a collection of core and exit routers. We also discuss a few challenges that arise with our approach and potential solutions for them. Finally, in closing, we argue that using data to inform FIB design is an interesting direction which can help in scaling with the growing demand for fast and in-expensive lookups, and is complimentary to existing approaches.

7 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their reviews and insightful comments. We are also grateful to P. Brighten Godfrey, Radhika Mittal, and Sambhav Satija for their valuable feedback on the paper draft.

REFERENCES

[1] S. Ashok, A. Partap, and A. Tahir. Optimizing memory usage for hybridfib design (https://drive.google.com/file/d/1O41f43pyhi2LH8g7Qdcep9ldWQbTVvJE/view?usp=sharing), 2022.

[2] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212, 2003.

[3] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[4] B. Indira, K. Valarmathi, and D. Devaraj. A trie based ip lookup approach for high performance router/switch. In *2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS)*, pages 1–6. IEEE, 2019.

[5] J. Lee and H. Lim. Binary search on trie levels with a bloom filter for longest prefix match. In *2014 IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, pages 38–43. IEEE, 2014.

[6] H. Lim and N. Lee. Survey and proposal on binary search algorithms for longest prefix match. *IEEE Communications Surveys & Tutorials*, 14(3):681–697, 2011.

[7] H. Lim, K. Lim, N. Lee, and K.-H. Park. On adding bloom filters to longest prefix matching algorithms. *IEEE Transactions on Computers*, 63(2):411–423, 2012.

[8] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[9] D. Shah and P. Gupta. Fast updating algorithms for tcam. *IEEE Micro*, 21(1):36–47, 2001.

[10] S. Shi and C. Qian. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(2), jun 2020.

[11] A. Shirmarz and M. Sabaei. Evaluation and comparison of binary trie base ip lookup algorithms with real edge router ip prefix dataset. *Evaluation*, 7(6), 2016.

[12] K. Vaidya, E. Knorr, M. Mitzenmacher, and T. Kraska. Partitioned learned bloom filters. In *International Conference on Learning Representations*, 2021.

[13] M. Wang and M. Zhou. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment*, 2019.

[14] L.-C. Wuu, T.-J. Liu, and K.-M. Chen. A longest prefix first search tree for ip lookup. *Computer Networks*, 51(12):3354–3367, 2007.

[15] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, page 313–324, New York, NY, USA, 2009. Association for Computing Machinery.

[16] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108, 2013.