CuVPP: Filter-based Longest Prefix Matching in Software Data Planes

Minseok Kwon†, Krishna Prasad Neupane†, John Marshall‡, M. Mustafa Rafique†
†Rochester Institute of Technology
‡Cisco Systems, Inc.
†{jmk, kpn3569, mrafique}@cs.rit.edu; ‡jwm@cisco.com

***Abstract*—Programmability in the data plane has become increasingly important as virtualization is introduced into networking and software-defined networking becomes more prevalent. Yet, the performance of programmable data planes on commodity hardware is a major concern, in light of ever-increasing network speed and routing table size. This paper focuses on IP lookup, specifically the longest prefix matching for IPv6 addresses, which is a major performance bottleneck in programmable switches. As a solution, the paper presents CuVPP, a programmable switch that uses packet batch processing and cache locality for both instructions and data by leveraging Vector Packet Processing (VPP). We thoroughly evaluate CuVPP with both real network traffic and file-based lookup on a commodity hardware server connected via 80 Gbps network links and compare its performance with the other popular approaches. Our evaluation shows that CuVPP can achieve up to 4.5 million lookups per second with real traffic, higher than the other trie- or filter-based lookup approaches, and scales well even when the routing table size grows to 2 million prefixes.**

***Index Terms*—IP Lookup; Packet Forwarding; Programmable Switches; Cuckoo Filters; Vector Packet Processing; Cache Locality**

I. Introduction

Programmability in network switches (or data planes) has become increasingly important with increasing network virtualization in the Internet infrastructure and large-scale data centers. As Software-Defined Networking (SDN) [1], Network Function Virtualization (NFV) [2], OpenFlow [3], and Open vSwitch [4] become popular for realizing network sharing, there have been attempts to improve programmability in switches, e.g., software switches [5], [6], domain-specific programming languages with P4 [7], PICSES [8], and in the Linux kernel with eBPF [9].

A critical challenge in data plane programmability is to maintain high-speed packet processing performance with ever-increasing link speed to hundreds of Gbps or Tbps. Another challenge is the rapid growing routing table size, e.g., more than 500,000 entries. The emergence of the Internet of Things (IoT), cloud computing, and IPv6 addressing further increases the routing table size, which is mitigated to some extent by leveraging prefix aggregation [10]. Moreover, software switches running on commodity hardware should be able to provide rapid address lookups ideally at line speed without the help of specialized hardware such as Ternary Content-Addressable Memory (TCAM).

Given these trends and challenges, this work is motivated by two observations. *First*, packet processing at software switches such as IP lookup can benefit from utilizing *cache locality* to the extreme. If packets are processed individually, limited cache locality for CPU instructions exists since each packet fetches a new instruction without reusing the old one. Exploiting locality in the data cache is challenging because network prefix lookup is incremental from long to short, not promoting the reuse of the old data in the cache. *Second*, the IP lookup process, specifically longest prefix matching, is a major bottleneck in the scalability and performance of programmable data planes, especially for Wide-Area Networks (WAN) [11], [12]. The prefix length is unknown for a given destination IP address, e.g., 32 prefixes for IPv4 and 128 for IPv6. Therefore, the lookup process is complex and computationally intensive, and increasing routing table size exacerbates the detrimental effect of longest prefix matching.

IP lookup challenge has been extensively studied in the past mainly with the focus on physical switches using TCAM [13], tries [11], [12], [14], and hashing with filters [15]. Recent efforts have studied software switches with specialized hardware, e.g., graphics processing unit (GPU) [5], resulting in extreme heat dissipation and power consumption. Alternate approaches, such as using algorithmic enhancement [16] is not scalable in efficiently utilizing cache memory with increasing routing table size.

We posit that high-speed longest prefix matching at software switches is achievable using *filter technologies* and *batch processing*. To this end, we propose a novel longest prefix matching algorithm, CuVPP, that is built on top of the FD.io Vector Packet Processing (VPP) framework [17]–[19] taking advantage of the cuckoo filter stored in the cache constantly as a pre-screening measure. We hypothesize that filter-based lookups are inherently well-suited to VPP and that they are better than using tries, especially for edge routers with more than 500K, and possibly millions in the future, prefixes in the Forward Information Base (FIB). The proposed CuVPP utilizes both instruction and data cache. Furthermore, the filter is compact to fit entirely in the cache in contrast to using a trie based approach where the trie grows as more prefixes are added with pointers consuming non-trivial space resulting in impeding data cache locality. Finally, new prefix insertion is easy with the filter that uses hashing, whereas insertion into the trie entails complicated structural changes. In a filter-based lookup approach, the prefixes that return positive from the filter screening need to be validated with the FIB due to possible false positives. Hence, reducing the number of false positives is critical to achieving a scalable and faster packet processing routing system.

We evaluate CuVPP with *real network traffic* using IPv6 prefixes from operational routers. Our results indicate that

978-1-7281-6677-3/20/$31.00 ©2020 IEEE

TABLE I: Number of CPU cycles per second and lookup rates (in MPPS) with VPP for a single entry lookup.

Number of Prefix Lengths	Random		Prefix Distribution	
	Number of Cycles	Lookup Rate	Number of Cycles	Lookup Rate
1	452	8.55	441	8.60
33	899	4.34	923	4.30
129	1,610	1.70	1,490	1.68

utilizing cache locality with the filter technology helps enhance the lookup rates by up to 24%, 51%, and 128% over Base VPP, VPP with Bloom filters, and VPP with a trie, respectively. We also demonstrate that the lookup rate of CuVPP scales in proportion to the number of processing threads.

Overall, this paper makes the following contributions:

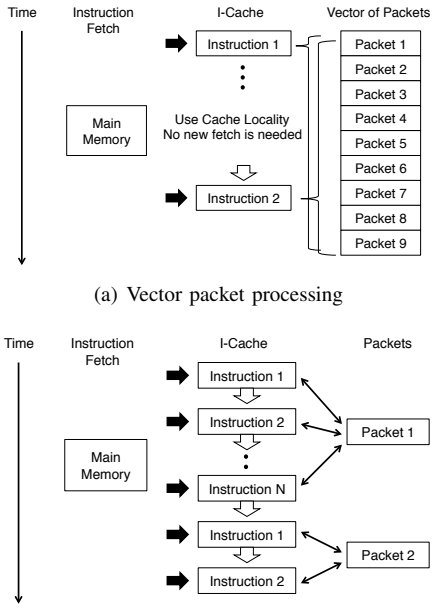
- We show that IP lookup performance can be significantly improved using **cuckoo filters** with **cache locality** with batch processing in the software switches.
- We develop a novel IP lookup approach, CuVPP, leveraging cuckoo filters, which is scalable with the increasing number of routing table entries. CuVPP improves the longest prefix matching performance that is a critical bottleneck in packet processing in the software switches.
- We implement CuVPP as part of **the real software switch VPP** and provide a comprehensive evaluation using popular alternative approaches with realistic data sets for network prefixes and traffic.

The rest of the paper is organized as follows. Section II provides a motivating use case, and Section III discusses the building blocks and enabling technologies. Section IV presents the main architecture, Section V discusses a comprehensive evaluation of CuVPP, and Section VI gives related work. Finally, Section VII concludes this paper.

II. Motivating Use Case

To understand the implication of longest prefix matching on packet forwarding performance, we conducted simple experiments that highlight the lookup performance. The testbed consists of one server that runs VPP and another server for traffic generation. The two servers are connected via an 80 Gbps network link. One thread runs on one core in the VPP server to process the incoming packets sent from the traffic generator at 10 MPPS (Million Packets Per Second). The complete configurations of our testbed are provided in Section V-D2. We compare the forwarding performance of three cases with extremely different search space sizes in the routing table. The first case has only one prefix in the table, and all the incoming addresses are matched to the prefix. This configuration is a replica of the Continuous System Integration and Testing (CSIT) baseline test with one core and one thread [20]. The second and third cases use 0-32 and 0-128 prefix lengths in the routing table that are matched to IPv4 and IPv6 addresses, respectively. The time complexity of the longest prefix matching can be understood by comparing the time taken for processing and forwarding packets in these cases.

We show the packet forwarding time with VPP in Table I measured in CPU cycles and lookup rates for both the random and prefix distribution traffic patterns. The single entry case uses the least CPU cycles yielding the highest lookup rates for both traffic patterns. As the only difference between the three cases is the number of prefix lengths to match in longest prefix matching, this result clearly demonstrates that more prefix lengths to evaluate require more processing cycles and lower lookup rates. This impact exacerbates with IPv6 addresses as the prefix lengths to probe increases from 1 to 129 requiring around 4× more CPU cycles for processing, reducing the lookup rate by 8× as compared to the single entry case.



(a) Vector packet processing

(b) Individual packet processing

Fig. 1: Illustration of vector packet processing versus individual packet processing.

III. Enabling Technologies

This section discusses the building blocks of CuVPP, i.e., vector packet processing, cuckoo filters for pre-screening, and filter-enabled longest prefix matching.

A. Vector Packet Processing (VPP)

VPP [17] enables high-throughput packet processing to extreme scale, e.g., 10 Gbps with 15 MPPS per core, in programmable switches using commodity hardware. It collectively processes the vector of packets to maximize the effect of cache locality as shown in Figure 1(a). Considering that the L1 instruction cache can determine whether a packet contains an IPv6 address, the first packet in the vector fetches the instruction into the cache, and the rest can be processed concurrently by fully utilizing the instruction cache. The size of the vector is determined dynamically, i.e., between 1 to 255, depending upon the packet arrival rate. If the packets are processed individually as shown in Figure 1(b), then each packet must be read into the cache independently. Moreover, the cost of packet processing in VPP is amortized over time leading to more stable and higher throughput even when

the number of packets fluctuates in the vector. VPP is a modular and extensible architecture allowing future hardware acceleration technologies to be easily adopted. Finally, VPP runs completely at the user level and provides easy-to-use high performance Application Programming Interfaces (APIs).

B. Cuckoo Filters

Cuckoo filter [21], similar to Bloom filter, efficiently test set-membership using extremely small space. It uses two hash functions to determine the insertion place of new items. If both places are unavailable, then one of the items is relocated to its alternative place using the other hash function, and the new item is inserted into the vacated place. This process continues if the alternative place is also occupied. This relocation process is called a *cuckoo move*, and it continues until all items are inserted or a predefined upper-bound for cuckoo moves is reached. The cuckoo filter supports dynamic insertion and deletion without increased space overhead, whereas Bloom filters do not allow item deletion. Counting Bloom filters [22] allow deletion with 4× higher space requirements.

C. Filter-enabled Longest Prefix Matching

In IP lookup, various techniques [15], [23], [24] can pre-screen network prefixes to limit candidates for longest prefix matching. A Bloom filter with prefix entries is placed in the on-chip memory while the FIB or forwarding table with full prefix entries are stored in the off-chip memory. The IP address of an incoming packet is concurrently tested against the entries in the filter to get a significantly small, e.g., 3-4, set of matching prefixes that are in turn tested against the forwarding table entries using hashing to eliminate the false positives. The performance gain mainly comes from filter testing as it is performed on the data located in the on-chip memory.

IV. CuVPP

This paper aims to show that an IP lookup algorithm with a cuckoo filter reduces lookup time, makes updates easy, and requires small memory space, and to demonstrate the critical role of caching in achieving high performance in software-based switches. Longest prefix matching requires all possible prefix lengths to be tested against entries in the FIB, which is the primary source of delay in the lookup time as shown in Section II. The computational cost further increases for the IPv6 addresses because of a longer prefix length, i.e., from 0 to 128. To address this challenge, we seek to leverage cache locality in conjunction with the batch packet processing. This enables prefix matching to only access cache in most cases, with reduced off-chip memory access for a given IP address. This approach entails the routing information to be compactly represented so that it can fit in the cache memory. Moreover, the data structure for the cache contents should be designed to support easy and rapid updates with minimal computational overhead. For faster processing, multiple threads are created for concurrent prefix lookups and updating the cache memory. Another critical challenge is to enable concurrency control for efficient cache memory accesses as tens of threads concurrently execute to provide extremely high lookup rates.

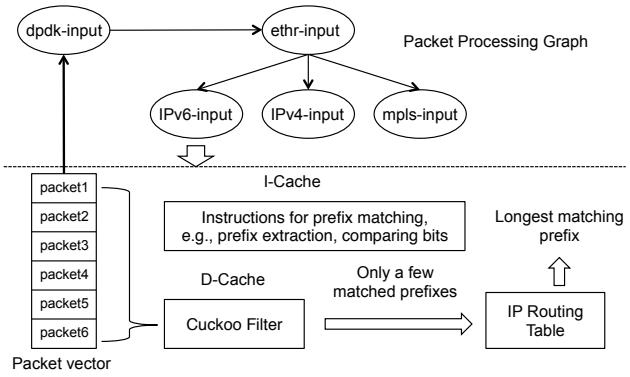


Fig. 2: Lookup engine architecture with filter and VPP technologies.

A. Architecture

The architecture of CuVPP is inspired by an early work in filter-based IP lookup [15] as shown in Figure 2. The main difference between CuVPP and the conventional filter-based IP lookup [15] lies in batch packet processing with the filter in the data cache (D-Cache) as packets go through the packet processing graph. The filter is used to pre-screen the prefixes that are not matched to a target IP address, which significantly reduces the number of lookups at the FIB in off-chip memory. Since the latency for off-chip memory access is considerably larger than the on-chip cache memory, this reduced number of lookups helps increase the lookup speed. It is critical to place the filter in the cache because all prefixes need to be tested at extremely high speed in the pre-screen stage. To fit contents in the cache, the size of the filter should be maintained to be compact. Note that lookups to the main memory still needs to be conducted despite the pre-screening by the filter to avoid false positives. Therefore, it is important to reduce the number of false positives to improve the overall lookup time.

The incoming packets in CuVPP are concurrently processed as a vector goes through the packet processing graph. Initially, the integrity of the packets is determined in order to pre-screen illegitimate ones according to defined rules, such as access control based on the source and destination addresses. Next, the packets are examined to determine the appropriate protocol depending on the destination address type, e.g., IPv4, IPv6, MPLS, and multicasting. The packets that are detected to use IPv6 are looked up in the filter for the longest prefix matching, i.e., all possible prefixes of 0 through 128 prefix lengths are tested against the filter, which returns a set of matched prefixes whose number is significantly less than the original search space. Those matched prefixes are tested against the FIB to eliminate false positives. Out of the final matched prefixes, the longest prefix is selected as the network prefix of an incoming packet, and the packet is forwarded to the next hop given by the prefix entry in the FIB. The performance can be improved by testing the prefixes from long to short sequences in FIB so that the process can be terminated as soon as a suitable prefix is matched.

The speedup from the VPP packet processing primarily emanates from utilizing the L1 instruction cache (I-Cache)

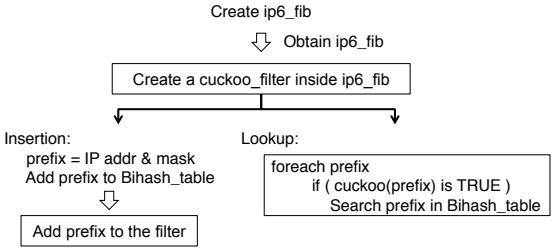


Fig. 3: Updated VPP code base for lookup and insertion supported by the cuckoo filter. New parts are represented within a box.

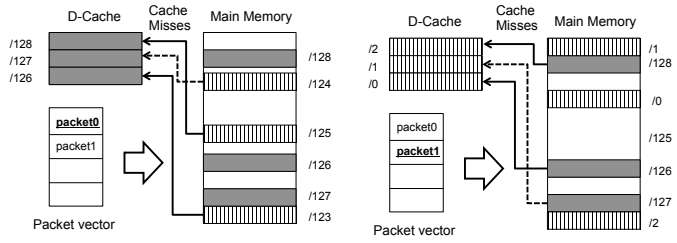


Fig. 4: Lack of temporal and spatial locality in D-Cache with a vector of packets.

[18], [20]. CuVPP focuses on the data cache (D-Cache) to show that further speedup is achievable by utilizing the D-Cache along with the I-Cache in VPP. To reduce the long delay incurred by conducting a complete comparison of all prefix lengths, a cuckoo filter is placed in D-Cache for extremely fast pre-screening of the prefix lengths. Conversely, in a trie-based lookup approach, a trie can also be placed in the D-Cache, however, the size of the trie grows more rapidly than the filter as more prefixes are inserted, and eventually, the entire trie cannot fit in the D-Cache.

B. Implementation

While VPP allows adding a new component as a plugin, we implemented CuVPP directly into the VPP core code base. Our changes mainly include adding the cuckoo filter, lookup and insertion mechanisms, and modifications in the VPP code base to support the new features. There are two reasons for directly implementing our approach in VPP instead of adding it as a plugin. First, it is complex to use a plugin as the filter interacts with the IPv6 and FIB table structures. Second, direct implementation in the VPP core ensures high performance by eliminating component interfacing and communication overheads. The filter implementation is placed on top of the FIB-based lookup and enforces only the prefixes that pass through the filter to perform FIB lookups. We have also added a set of command-line interface (CLI) commands to show additional statistical data, e.g., the number of FIB lookups, the number of cuckoo filter lookups, filter size, CPU clock cycles, and the lookup time.

Figure 3 shows our modifications to the lookup mechanism in the VPP system. In the first step, we create a cuckoo filter and add it to `ip6_fib`. In the second step, we modify the lookup module, so that the membership of each prefix is probed through the filter, and only the prefixes that the filter has been searched in the FIB hash table. For hashing, we use CRC-32 with bounded-index extensible hashing (bihash) that uses two-level lookup tables [25]. The insertion module requires adding prefixes to the filter as well as the hash table. The time complexity of hash computation is critical to reduce the overall lookup time as a non-trivial number of prefixes are hashed and tested for both the filter and hash table. To this end, we use 16 bits for a fingerprint generated by CRC-32, which is significantly faster than the cryptographic hash functions such as MD5 [26] and SHA-1 [27]. Ideally, the filter should be resident in the cache all the time to maximize the cache locality. A common technique that we adopt is to frequently access the filter. In doing so, it is crucial to have a compact filter size and have the memory page size defined such that the entire filter can fit in the cache. There are processor-assisted techniques, e.g., Intel Cache Allocation Technology (CAT) [28], that allows a process to reserve cache space. We did not use such approaches as they are more relevant for multiple processes competing for the same cache space.

C. Cache Locality with Batch Processing

VPP processes incoming packets in batches to significantly reduce the processing time by exploiting I-Cache locality. However, it fails to utilize D-Cache locality as shown in Figure 4 where longest prefix matching is performed for packets 0 and 1. We focus on the cache locality for processing a single packet and during the transition between packets. Without loss of generality, we assume that D-Cache can have up to three prefixes. To find the longest prefix quickly, the prefixes in the routing table are searched from long to short, that is, /128 to /0. Considering that packet 0 is tested for longest prefix matching, and the prefixes /128, /127, and /126 are tested, and no match is found, these prefixes are brought into D-Cache as they have been accessed. Next packet 0 is tested for the prefixes /125, /124, and /123 resulting in cache misses as none of the prefixes in D-Cache are reused by the subsequent lookups.

Another interesting observation with cache utilization comes from the transition from packets 0 to 1. When the search for packet 0 ends, D-Cache has the prefixes for /2, /1, /0 as shown in Figure 4. Now longest prefix matching for packet 1 begins, starting from the prefixes /128, /127, and /126. However, none of the prefixes residing in the D-Cache are useful and they cause cache misses as no temporal locality exists in the search. Furthermore, there is no spatial locality as each prefix is mapped to the routing table by hashing, which is not sequentially accessed. Also, dynamic allocation in the main memory further increases random memory block accesses and reduces spatial locality. Note that the cache locality cannot be exploited if the packets are processed individually rather than in a vector. A packet accesses different data in the main memory and moves them into D-Cache as it goes through different nodes in the packet processing graph (Figure 2). When the next packet begins its processing, the target data

block cannot be found in the D-Cache and causes a cache miss.

To promote the locality of D-Cache, CuVPP creates a cuckoo filter with only membership information to determine whether there is a match and keeps this filter in the D-Cache. The filter contains only membership information and is compact enough to fit in the D-Cache. This way, we artificially create temporal and spatial locality so that the longest matched prefix is always found in the D-Cache. This gives partial locality as the filter contains compressed membership information and the complete prefix information can be obtained by accessing the main memory. The cost of main memory access is amortized by the benefit, i.e., significantly reduced search time by using a filter in D-Cache.

D. Updating FIB and Filter Entries

While the lookup performance is critical for high-performance forwarding, the performance of update operations is equally important as any delay or errors in the update operation may result in packet forwarding to suboptimal routes or even packet losses because of increased packet processing time. As described in Section III-B, the fingerprint of a new prefix is inserted into one of the two positions that are computed by a hash function combined with the XOR operation in the filter. A prefix can be deleted by removing a matched fingerprint from one of those two positions. Deletion operation is extremely simple and fast requiring only two memory accesses for those two positions. It is even faster for access to CuVPP cache.

Insertion may take longer if both the positions are already occupied triggering cuckoo moves, i.e., displacing a prefix from one of the two positions and moving to the alternate position, and the process continues until all the prefixes are inserted [21]. Interestingly, the number of cuckoo moves (or a cuckoo path) is bounded in insertion when filter occupancy is not high. The number of cuckoo moves is $O(1)$ on average and $O(log\ n)$ in the worst case with high probability, where n is the number of prefixes if the load factor is 50% or lower [29]. If more hash functions are used (e.g., ≥ 2), a cuckoo path of $O(1)$ is possible even when the load factor is greater than 50% [21]. With two hash functions, it is important to keep filter size sufficiently large so that the load factor remains less than 50%. Keeping the load factor low is challenging because more devices need IP addresses and the number of prefixes increases rapidly leading to FIB explosion. We use two hash functions with 6 MB filter size that can hold up to 3 million entries. We can use more hash functions at the cost of increased computational complexity to increase the filter utilization without significantly increasing the filter size while achieving $O(1)$ cuckoo moves and low insertion delay.

Insertion and deletion in the cuckoo filter are more efficient than tries. For adding a new prefix, a target node is located in the trie as the parent, visiting all the nodes on the path from the root to the parent node resulting in the equal number of memory accesses as the prefix length. This number of memory accesses is from 0 to 32 for IPv4, and of course, the number increases drastically for IPv6 that has prefix length from 0 to 128. In contrast, updates in the cuckoo filter are agnostic to the number of prefix lengths because only two hashes are computed for either IPv4 or IPv6 addresses. For deletion, the trie requires up to the prefix length memory accesses to locate the node that contains the next hop of the target prefix. It takes only two memory accesses in the filter to probe the fingerprints at the two positions of the target prefix and remove one matched fingerprint.

While the FIB in off-chip memory is being updated, no lookup operation can be conducted in the FIB, degrading lookup performance. One approach to tackle this challenge is to keep two copies of the FIB, one for lookup and the other for update operations. When the update operation is complete, the FIB copy for lookup is simply replaced with the FIB that has been updated. Moreover, multiple threads can access the filter for lookup and update operations concurrently requiring implementing threads synchronization mechanisms. Techniques, such as, optimistic cuckoo hashing [30] and locking after discovering a cuckoo path [31], can be applied to further improve update performance.

E. Multi-Threaded Lookup

VPP has single- and multi-worker modes, and the packet processing can be further improved by using multiple threads running on multiple cores. All of the functions in the packet processing are read-only except prefix insertion into the routing table and filter. We believe that the speedup should be close to linear with an increasing number of threads. One possible bottleneck may occur at the receiver queue at the NIC from which threads read incoming packets. If the scheduling mechanism is not well-designed to distribute the packets to multiple cores efficiently, some threads may stall for packets resulting in reduced overall packet processing performance. We evaluate the impact of an increasing number of threads in the evaluation section, however, the scheduling mechanism is kept as the future work for our system. CuVPP uses no extra memory for multi-threading since all threads share the filter and the routing table in the main memory. Furthermore, all shared data structures in the multi-worker mode must be implemented in a thread-safe fashion. For instance, a static counter or a flag variable shared by multiple threads degrade performance significantly causing synchronization issues. Instead, a vector of variables can be used so that each thread can access an individual thread-safe variable with no synchronization overhead. We implemented several custom VPP CLI commands, and experienced performance degradation for more threads due to static non thread-safe variables.

V. Evaluation

Our evaluation focuses on evaluating and analyzing the CuVPP lookup algorithm as a standalone entity with minimal interactions with VPP as well as in the context of VPP. Our implementation of CuVPP is publicly available[1].

[1] https://github.com/cuckoovpp/CuckooVPP

A. Datasets

1) Network Prefixes in the FIB: We use two datasets from routers in real networks for evaluating the proposed system. The first dataset is from RouteViews [32], specifically a LINX BGP archive for IPv6 (named RV-linx) with a total of 20,440 prefixes. The prefix lengths in RV-linx are not uniformly distributed as only 34 out of 128 prefix lengths have at least one prefix, and two prefix lengths, i.e., /48 and /32, account for more than 73% prefixes, and the top six prefix lengths including /40, /44, /36, and /29 account for over 90% of all the prefixes. We scale prefixes following this prefix length distribution to 500K, 1M, 1.5M, and 2M prefixes to evaluate the scalability of CuVPP. The second dataset is from a Deutsche Telekom Looking Glass server (named DT-IP6) that has 513 prefixes of 17 different prefix lengths [33]. The prefix lengths in DT-IP6 are non-uniformly distributed where the majority (over 72%) is /48, and there are few /67, /17, and /16 prefixes. We scale the prefixes for large-scale experiments following this prefix length distribution. We note that not much dataset is publicly available for IPv6 prefixes mainly because IPv6 is still not publicly deployed as widely as IPv4.

2) Network Traffic Patterns: In IP lookup, incoming network traffic patterns have an impact on which network prefixes are searched for matching, and thus impact the overall search time. We conjecture that filter-based approaches including CuVPP, in contrast to tries, are agnostic to the traffic patterns since all prefixes must be tested by the filter. To this end, we use two different traffic patterns. The IP addresses are generated either at *random* out of 2^{128} values or following the *prefix distribution*, either RV-linx or DT-IP6, in the routing table. An IP address is produced by bitwise-ORing a prefix and the rest of the address is created at random. Prefix distributions seemingly have performance implications on IP lookup mechanisms. A study of such implications, however, is beyond the scope of this paper, and thus, we leave an exploration of this avenue for future work.

B. Data Planes for IP Lookup

We evaluate CuVPP in two dimensions in the VPP realm using other flavors of filters and with a trie. For obtaining base VPP performance, we use stock VPP v17.07-rc0. VPP with a trie leverages a two-way trie as the pre-processing unit to narrow down the search space for prefix matching. The performance of VPP with a trie reflects packet processing time in switches more accurately. It uses hashing for IP lookup in which all the prefix lengths are tried until the longest prefix is found. The other two approaches used in our evaluation are CuVPP and VPP with a Bloom filter.

C. Performance Metrics

We use lookup rate, memory access time, memory size, and time for updates as the performance metrics in our evaluation. Lookup rates indicate the speed of routing table lookup, and is defined as the number of lookups performed per second in Million Packets Per Second (MPPS) or Million Lookups Per Second (MLPS). MPPS and MLPS are not the same as each packet may take more than one lookup or different numbers of lookup. We measure the average memory latency per lookup to understand read and write overheads of memory accesses. In our evaluation, memory size indicates the amount of memory consumed by the routing table, filter, and trie, both in the cache and main memory, for the proper functioning of a particular approach. Time for updates measures the overall time taken to insert a prefix to the forwarding systems, e.g., hash tables, filters, or tries.

In the measurements for these metrics, we vary routing table size, prefix distribution in the table, traffic patterns, the number of worker threads, and the load of the filter or trie, as controllable parameters. The routing table size is the number of entries in the lookup table, and the prefix distribution is the distribution of prefix length in different IPv6 datasets. The traffic patterns represent the distribution of IPv6 addresses in arriving packets, and we use random and prefix-compliant traffic as discussed in the previous section. We change the number of worker threads to evaluate the effect of multi-threading, and the load of the filter or trie to understand its impact on the prefix insertion operations.

D. Test Configurations

We evaluate the performance of the lookup algorithms in both a synthetic lookup-only setting and a real testbed configuration for a comprehensive evaluation of CuVPP. We now present the details of our test configurations before presenting the performance evaluation.

1) **Lookup-Only Setting:** We create an isolated environment to focus on the lookup performance in a conventional setting [11]. Each lookup mechanism reads IP addresses from a file, searches the routing table with an address, and finds the next hop. The mechanism runs standalone using a single thread with negligible interference from other processes. We use a Cisco UCS C240 M4 Rack Server with an Intel Xeon 2.4 GHz E5-2640 v4 processor with dual 10 cores (hyper-threading disabled), and 25 MB (L3), 256 KB (L2) cache, and 64 GB DDR memory running Ubuntu 14.04.05 for this configuration. Note that the filter can fit in the L3 cache because the size of both the Bloom and Cuckoo filters is 6 MB, while the trie cannot as it requires 581 MB (see Section V-H) for storage. In each experiment, we process 500,000 packets for lookup.

2) **Real Network Testbed:** We configure a real network testbed with a traffic generator to evaluate our system. The testbed consists of two Cisco UCS C240 M4 Rack Servers with the same specifications as the lookup-only setting. The two servers are connected via 80 Gbps fiber optic cable with two Intel XL80 dual-port 40G QSFP plus network interface cards (NICs). One server generates and sends traffic to the other server for processing. The testbed is isolated from the external network with no generated traffic traveling outside the testbed network, however, the control traffic can be transmitted from a host on the external network for management and configuration. We use TRex [34] for the traffic generation, which is a low-cost packet generator from Cisco using Intel

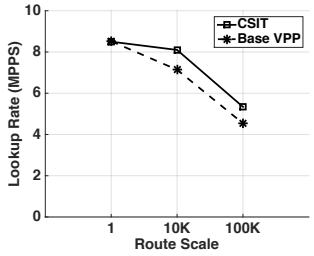
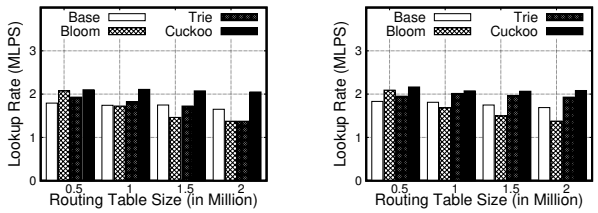


Fig. 5: Lookup rates comparison for CSIT benchmark and Base VPP on the real network testbed. Route scale is equivalent to routing table size.



(a) Random network traffic (b) Prefix-compliant network traffic

Fig. 6: Lookup rates for RV-linx. To save space, we omit VPP for each lookup algorithm, e.g., Base means Base VPP, and Cuckoo means CuVPP. Also, MLPS stands for Million Lookups Per Second.

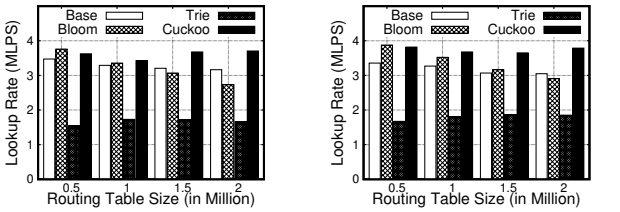
DPDK [19] for fast packet processing and can send traffic up to 200 Gbps.

To ensure the correctness of our results, we compare the VPP installed on the testbed against the VPP in the CSIT testing environment [20]. CSIT is a Cisco project for automated system performance testing and integration with VPP using TRex as the traffic generator. We replicate a basic experiment with one core and one thread conducted by CSIT for VPP performance, and achieve comparable performance as shown in Figure 5. We achieved a non-drop rate (NDR) of more than 8 MPPS, which is comparable to the CSIT performance [20].

E. Lookup Rates: The Lookup-Only Setting

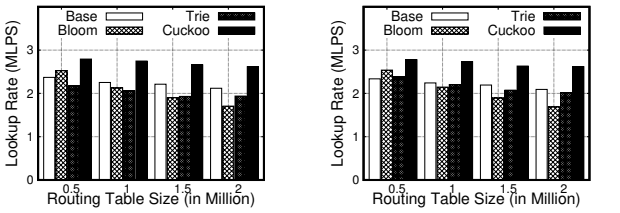
Figure 6 and Figure 7 show lookup rates performance for RV-linx and DT-IP6 datasets, respectively. The x-axis denotes routing table size, and the y-axis denotes the achieved lookup rates measured in MLPS. With the RV-linx dataset, CuVPP improves the lookup rates by approximately 18–23% over Base VPP approach, and by 3–51% over VPP with a Bloom filter approach, and by 7–10% over VPP with a trie approach. The improvements are 13–24%, 4–30%, and 104–128% over Base VPP, VPP with a Bloom filter, and VPP with a trie, respectively, for the DT-IP6 dataset. The observed improvements are similar for both the random and prefix-compliant traffic patterns. For the DT traffic, we observe similar trends except that the improvement of VPP with a cuckoo is significantly larger than VPP with a trie approach.

These results unveil two interesting observations. First, CuVPP has higher lookup rates than Base VPP, VPP with a Bloom filter, and VPP with a trie approach. The higher lookup rates of CuVPP are achieved mainly from reduced memory-access latency as it efficiently utilizes the cache memory as



(a) Random network traffic (b) Prefix-compliant network traffic

Fig. 7: Lookup rates for DT-IP6.



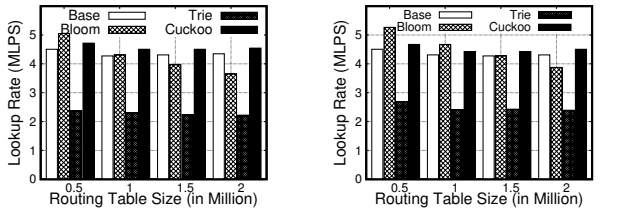
(a) Random network traffic (b) Prefix-compliant network traffic

Fig. 8: Lookup rates for the real testbed (RV-linx).

compared to the other approaches. Second, the lookup rates of CuVPP are consistent regardless of routing table size, but the lookup rates of the other three algorithms constantly decrease as the size of the routing table increases. This means that CuVPP scales well, and the lookup performance is not affected as long as the entire filter can fit in the cache memory. The search time in the trie slightly increases when more prefixes are added because newly-added prefixes often overlap with the already-created paths in the trie and the height of the trie is bounded by 128. The high false positive rates affect the steady decrease of VPP with a Bloom filter in lookup rates. The space utilization of the Bloom filter is not as high as the cuckoo filter, e.g., the Bloom filter minimizes the false positive rate when half of the filter is filled, whereas the cuckoo filter can use almost 95% of its space with little impact on the false positive rate [21]. In our evaluation, the Bloom filter size is constrained by the available cache space leading to high false positive rates.

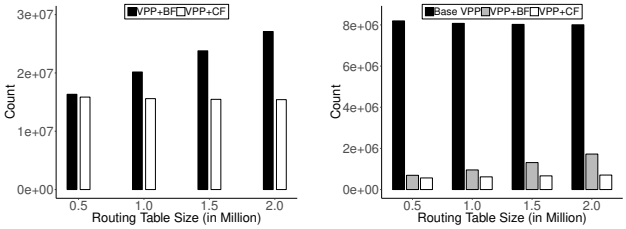
F. Lookup Rates: The Real Network Testbed

The lookup rate results obtained from the real network testbed are shown in Figure 8 and Figure 9. The datasets used to generate the prefixes in the routing table are the same as the lookup-only setting, namely RouteViews and DT RIB. For RV-linx, CuVPP shows the lookup rates 18–24% higher than the Base VPP approach, and 10–55% higher than VPP with a Bloom filter approach, and 15–35% higher than VPP with a trie approach, for both the random and prefix-compliant traffic. These improvement rates change to 4–5% as compared to the Base VPP approach, and about 4–25% against VPP with a Bloom filter approach, and 75–100% compared to VPP with a trie approach, for the DT-IP6 dataset. These results are overall consistent with those in the lookup-only setting, for similar reasons as explained in Section V-E. In our experiment, we found that both file and real traffic lookups have nearly the



(a) Random network traffic (b) Prefix-compliant network traffic

Fig. 9: Lookup rates for the real testbed (DT-IP6).



(a) Filter lookup (b) FIB lookup

Fig. 10: Profiles of lookup (DT-IP6).

same range of lookup time for a single lookup. Specifically, 98% lookups fall into a range of one ten-millionth (1E-07) seconds for both file and real traffic.

G. Lookup Profiling

We profile the routing lookups as a micro-architectural analysis for computing cost. Specifically, the profiles show the number of on-chip and off-chip memory accesses for filter and FIB lookups, respectively, comparing the four different lookup approaches. These profiles are obtained from the same testbed using the same datasets for the prefix distribution in the routing table and traffic patterns as the lookup rate measurement. In VPP, the ip6-input process takes the destination IP address and other fields from a packet, and then passes the packet to a respective process including ip6-lookup, ip6-local, ip6-icmp-input, among other processes. The ip6-lookup process is only responsible for the lookup part, therefore, we only measure lookup count in the ip6-lookup process.

Figure 10(a) shows profiles of filter lookup for both Bloom and cuckoo filters on different routing table sizes when 500,000 IP addresses are searched for the longest matching prefix. We use the prefix-compliant traffic, and the filter lookup begins from the longest, i.e., 128 bits, and continues in decreasing length until a matching prefix is found. The results show that as routing table size increases, more lookups occur in the Bloom filter, whereas the number of cuckoo filter lookups remain nearly the same. With 2M route entries, the number of filter lookups is reduced by nearly 90% in VPP with a cuckoo filter as compared to VPP with a Bloom filter. The increase in routing table size sets more bits in the Bloom filter, causing more false positives and ultimately leading to more filter lookups. Another contributing factor is the higher number of hash computations for the Bloom filter compared to the cuckoo filter that requires only two hash computations.

Figure 10(b) shows the number of routing table lookups obtained from the same experiment. It shows that the base VPP approach performs many unnecessary routing table lookups, unlike the filter-based lookup techniques. VPP with both the Bloom and cuckoo filters significantly reduces these unnecessary lookups, but the number of lookups in the Bloom filter increases with the size of the routing table. Overall, the cuckoo filter technique helps reduce the number of lookups for both filter and FIB lookup approaches.

The access to cache memory takes approximately 2–90× less time than the access to main memory depending on the cache types (L1, L2, and L3) [35]. We study the number of lookups for the filter and FIB, and their relations to memory access time in order to understand the effects of these different cache access times on the performance of the longest prefix matching. In our experiments, a total of 500,000 prefixes are inserted in the FIB following the prefix distribution of RV-linx, and the traffic pattern follows the same prefix distribution. We test a total of 500,000 IP addresses in the lookup-only setting.

TABLE II: Number of lookups for the filter and FIB, and latency for memory access for those lookups, i.e., the total time to perform all the given lookups.

Data Plane	Number of Lookups		Time (sec.)
	Filter	FIB	
Base VPP	0	16,987,150	0.274702
VPP + Bloom	16,987,150	1,205,520	0.240861
VPP + Trie	0	500,000	0.278208
CuVPP	16,987,150	648,800	0.219023

Table II shows the number of lookups performed for both the filter and the FIB table, and the total time taken for all the lookups. These results highlight several important observations. First, all the lookups in Base VPP approach are performed in the FIB as expected as it uses no filter. Second, VPP+Bloom and CuVPP approaches have the same number of filter lookups, which is equal to the number of FIB lookups in the Base VPP approach. These results show that both these algorithms perform lookups in the filter before accessing FIB. Third, CuVPP has significantly lower, i.e., almost 50%, FIB lookups compared to the VPP+Bloom technique despite the same number of filter lookups. The number of FIB lookups depends on the false positive rates, and the observed result implies that CuVPP has lower false positive rates than the VPP+Bloom approach. The number of FIB lookups in CuVPP is slightly higher than 500,000 including 500,000 IP addresses being looked up and 148,800 false positives. Fourth, CuVPP yields the lowest memory access time. This metric represents the latency required for communication between memory devices and the CPU. This overhead includes time taken for a lookup as well as hash computation. CuVPP requires more hash computations for computing fingerprints, however, it reduces overall lookup time by exploiting the cache locality and reduces the overall false positive rates.

Additionally, we measure cache miss ratios in the same experiments and report our observations in Table III. Our measurements show that the cache miss ratio of CuVPP is

TABLE III: Cache miss ratios for different lookup algorithms.

Data Plane	Cache References	Cache Misses	Cache Miss Ratio
Base VPP	209,668,571	13,921,490	6.640%
VPP + Bloom	240,289,331	14,692,333	6.114%
VPP + Trie	176,937,686	4,805,587	2.716%
CuVPP	166,405,389	324,970	0.195%

TABLE IV: Memory footprints for different data planes.

Data Plane	Memory Used (MB)
Base VPP	4,000
VPP + Bloom	4,006
VPP + Trie	4,581
CuVPP	4,006

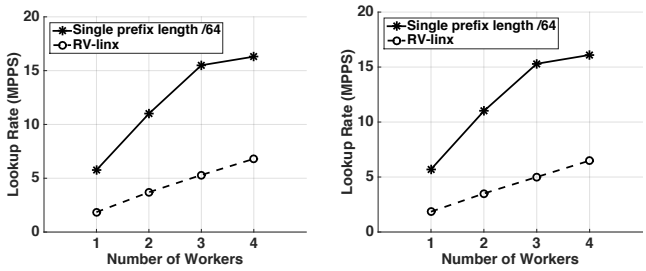
lower than the VPP+trie technique by an order of magnitude, and it is lower than Base VPP and VPP+Bloom approaches by six orders of magnitude. We note that CuVPP significantly reduces the cache miss ratio as compared to the other techniques studied in our evaluation.

H. Memory Footprint

We measured the memory (RAM) consumed for each data plane and report our observations in Table IV. The memory footprint is measured after inserting one million prefixes. Base VPP uses a total of 4 GB RAM. Bloom filter consumed 4.008 GB in total with 6 MB for storing the Bloom filter itself as a bit vector and the rest for storing other VPP modules. For the Bloom filter, there is a trade-off between the memory size that the filter uses and the false positive rate, and the false positive rates can be reduced for larger filter size at the cost of consuming more memory space. Therefore, the size of the Bloom filter is determined considering available memory, e.g., cache size, and the target false positive rate. The cuckoo filter has a total of 3M entries, and the size of each fingerprint is 16 bits. Hence, the size of the filter is 6 MB. This trade-off also exists for the cuckoo filter, but it requires less memory because of efficient space utilization as compared to the Bloom filters. We compare the memory footprint of VPP with a trie when the same prefixes are added, and found that the trie creates a total of 24,212,583 nodes each of which has two pointers of 8 bytes, the word length, and uses another 8 bytes for a boolean variable. Each node then uses 24 bytes resulting in approximately 581 MB (24,212,583×24B). Our results show that VPP with a trie uses significantly more main and cache memories than all the other three filter-based VPP data planes.

I. Scalability with Receive Side Scaling

We also evaluated CuVPP to observe if the lookup rate increases with the increasing number of threads. The Intel Network Adapter in our setup provides Receive Side Scaling (RSS) mechanism that enables packet processing to be shared across multiple processors or cores [36]. This enables better system cache utilization. Increasing the number of rx-queues is instrumental to maximize the RSS benefits. VPP allows specifying the number of rx-queues and tx-queues in its DPDK configuration. However, if RSS is disabled, then several workers process incoming packets from a single receive queue resulting in reduced overall performance.



(a) 10,000 routes (b) 100,000 routes

Fig. 11: Throughput increases as the number of workers in CuVPP increases from 1 to 4.

Figure 11 shows packet throughput with increasing numbers of workers and rx-queues. The testbed uses a single NIC that limits the benefits of multiple rx-queues after 16 MPPS throughput, which is close to the I/O bandwidth limit of the NIC in our setup. The experiment uses up to four workers (or threads), and the number of workers is kept equal to the number of rx-queues to avoid packet input bottleneck. To highlight the effect of the longest prefix matching, we evaluate two sets of routing table configurations, one with a single prefix length, namely /64, and the other based on the RouteViews dataset (RV-linx) [32] with a distribution of 33 prefix lengths in which most prefixes are either /32 or /48, for 10,000 and 100,000 routes in the table. The observed results show a nearly twofold increase in the packet throughput when two workers are used, and slightly less but steady increases for three and four workers. Furthermore, lookup throughput with single prefix length is 3–4× higher than those for the 33 prefix lengths due to the additional time taken for the longest prefix matching.

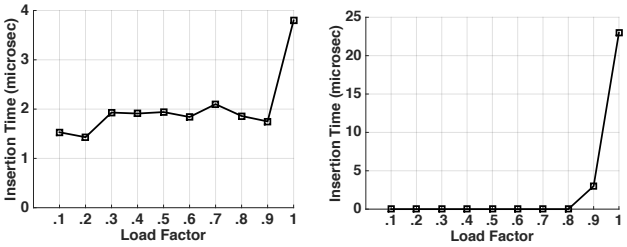
J. Updates

To evaluate our hypothesis on prefix insertion (Section IV-D), we measure the time taken for prefix insertion for the cuckoo filter. We also measure the number of cuckoo moves per insertion to validate that more cuckoo moves further delay the insertion process. Figure 12 shows that the insertion time and the number of cuckoo moves increase as the number of prefixes in the filter increases from 0 to 1M, and hence validates our conjecture. The number of cuckoo moves stays 0 until the load factor of 0.8, then goes up slightly for 0.9, and abruptly peaks when the filter becomes full. The number of cuckoo moves increases because more entries lead to hash collisions resulting in a higher insertion time.

VI. Related Work

A. Software Data Planes

VPP [17] is an open-source software switch that provides packet processing stack for routing and switching. Its design is based on the packet processing graph, which is inherently modular and extensible, such that new plugin nodes can



(a) Time for prefix insertion (b) Number of cuckoo moves

Fig. 12: Time taken for prefix insertion and the number of cuckoo moves with increasing load factor for a maximum of 1M entries.

be easily added. The key to its high performance lies with processing packets as a vector instead of individually to maximize the benefits of caching. Open vSwitch (OVS) [4] is arguably the most popular software switch that runs on top of hypervisor platforms to support virtual networking. OVS allows distributed virtual switching across multiple physical servers creating an illusion of one large server that virtual machines can utilize transparently. Other software switches, such as PacketShader [5], Switch-Blade [37], RouteBricks [38], and CuckooSwitch [6], run on commodity processors, e.g., GPUs or FPGAs, and provide limited programmability. CuckooSwitch is similar to our work such that it uses cuckoo hashing in the cache memory, and can achieve extremely high-speed lookup, e.g., 80 Gbps with 16 cores at 2.7 GHz. However, CuckooSwitch is designed for lookups in the LAN environment and does not require longest prefix matching.

B. P4-enabled Software Switches

P4 [7] is a programming language that facilitates building protocol-independent packet forwarding engines. The design of P4 is motivated by the limitations of extending OpenFlow to support more packet processing logic in SDN environments. P4 enables programmers to easily reconfigure packet processing rules at the switch. Moreover, P4 is agnostic to specific protocols and hardware architectures, making P4 versatile for different protocols and hardware platforms. However, it requires expert knowledge to upgrade, customize, or extend their packet processing mechanisms. PISCES [8], a programmable switch independent of specific protocols, tackles this challenge by using P4 as a vehicle to customize the behavior of software switches. In PISCES, a P4 program that defines packet processing rules is first translated into an intermediate representation, which in turn is optimized and compiled to C code to overwrites the parse, match, and action code in OVS. PVPP [39] is a similar effort as PISCES for VPP, with the focus on fine-tuning the functionalities of the switch to achieve higher performance. The use of VPP provides fine-grained access to low-level features, such as metadata access, pointer dereferencing, and match-action tables.

C. Routing Table Lookup

IP address lookup has been a challenge in high-speed packet processing at routers. The network prefix length is unknown due to the use of Classless Inter-Domain Routing (CIDR) requiring to inspect all matched prefixes in the table to find the longest matching prefix. High-speed IP lookup is even more challenging with increasing link speed at core routers. TCAM has been a popular technology for IP lookup due to its ability to inspect all prefixes with a single memory access at very high speed [13], [40]. However, TCAM has a high cost-to-density ratio and requires high power consumption. Another effort on IP lookup uses the trie data structure [41], [42] for searching the longest matching prefix along its structure from the root to a leaf. The primary limitations of trie-based lookup techniques, e.g., Lulea algorithm [43], Controlled Prefix Expansion [44], [45], LC-trie [46], Tree Bitmap [12], SAIL [14], and Poptrie [11], include high number of memory access, high update complexity, and large data structures exceeding the cache size as the number of prefixes increases. DXR [16] is another data structure that facilitates search using prefix ranges, instead of prefixes, to utilize cache efficiently. The effort that is closely related to the contributions of this paper is the early work in filter-based lookups and packet inspection and classification [15], [23], [47], however, it lacks in accessing the Bloom filter in parallel, which we overcome by maximizing cache utilization with vector processing in VPP. Recently, the work that comes close to this paper is MicroCuckoo [48] in which cuckoo hashing utilizes parallel data processing capabilities of FPGA for even faster IP lookup. One hurdle here is that FPGA is not yet as available as commodity hardware like x86 CPUs, especially in data centers.

VII. Conclusions and Future Work

This paper has presented a novel longest prefix matching algorithm in the Vector Packet Processing (VPP) programmable data plane, called CuVPP, using a cuckoo filter in the cache as the fast pre-screening mechanism. With highly-efficient batched packet processing that exploits instruction cache locality, CuVPP utilizes data cache locality to minimize the number of lookups to the off-chip memory resulting in significantly reduced lookup time and improved router performance. CuVPP achieves higher lookup rates than other popular approaches, specifically, Base VPP, VPP with a Bloom filter, and VPP with a trie, in the range from 5% to more than 100%, depending on the number of prefixes, the number of prefix lengths, and the traffic patterns. In the future, we would like to extend CuVPP with general matching capabilities beyond longest prefix matching such as TCAM-style exact matching, which has more use cases, e.g., ACLs or NIDS. Virtual switches in data centers could also benefit from this CuVPP-based exact matching as the translation between virtual and physical IP addresses needs exact matching, not longest prefix matching. We are also interested in further exploring the interplay between batch processing and cache with varied batch size, cache size, and cache eviction algorithms. Finally, we are interested in exploring the programmability aspect in relation to CuVPP, specifically designing some programmable constructs that enable an easy configuration of filter-based lookup with VPP.

REFERENCES

[1] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[2] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

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

[5] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *ACM SIGCOMM*, 2010.

[6] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, High Performance Ethernet Forwarding with CuckooSwitch," in *ACM CoNEXT*, 2013.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

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

[9] I. V. Project, "eBPF," 2019, https://www.iovisor.org/technology/ebpf.

[10] G. Huston, "CIDR REPORT," potaroo.net, https://www.cidr-report.org/as2.0/.

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

[12] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, Apr. 2004.

[13] A. J. Mcauley and P. Francis, "Fast Routing Table Lookup Using CAMs," in *IEEE INFOCOM*, 1993.

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

[15] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *ACM SIGCOMM*, 2003.

[16] M. Zec, L. Rizzo, and M. Mikuc, "DXR: Towards a Billion Routing Lookups Per Second in Software," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 29–36, Sep. 2012.

[17] FD.io, "VPP (Vector Packet Processing)," May 2017, https://fd.io.

[18] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, D. Rossi, and J. Tollet, "Batched packet processing for high-speed software data plane functions," in *IEEE INFOCOM INFOCOM WKSHPS*, 2018.

[19] D. Project, "Developer Quick Start Guide," 2019, https://www.dpdk.org/.

[20] C. Systems, "FD.io CSIT (Continuous System and Integration Testing)," 2017, https://wiki.fd.io/view/CSIT.

[21] B. Fan, D. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *ACM CoNEXT*, 2014.

[22] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 281–293, 2000.

[23] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," in *IEEE Micro*, 2003.

[24] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 1, pp. 17–34, 2006.

[25] D. Barach and B. Batacharia, "Bounded index extensible hash-based IPv6 address lookup method," 2008, uS Patent US7325059B2.

[26] R. L. Rivest, "The md5 message-digest algorithm," Internet Requests for Comments, RFC Editor, RFC 1321, April 1992, http://www.rfc-editor.org/rfc/rfc1321.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1321.txt

[27] D. Eastlake and P. Jones, "Us secure hash algorithm 1 (sha1)," Internet Requests for Comments, RFC Editor, RFC 3174, September 2001, http://www.rfc-editor.org/rfc/rfc3174.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3174.txt

[28] Intel, "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family," 2016.

[29] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004.

[30] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *USENIX NSDI*, 2013.

[31] X. Li, D. G. Andersen, M. Kaminsky, and M. Freedman, "Algorithmic Improvements for Fast Concurrent Cuckoo Hashing," in *ACM EuroSys*, 2014.

[32] Advanced Network Technology Center at the University of Oregon, "The Route Views Project," May 2017, http://www.routeviews.org/.

[33] B. L. Glass, "BGP: the Border Gateway Protocol Advanced Internet Routing Resources," 2017, http://www.bgp4.as/looking-glasses.

[34] C. Systems, "TRex – Realistic Traffic Generator," 2015, https://trex-tgn.cisco.com/.

[35] D. Levinthal, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors," Intel, Tech. Rep., 2009.

[36] Intel, "Receive Side Scaling on Intel Network Adapters," Sep. 2017.

[37] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster, "Switch-Blade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware," in *ACM SIGCOMM*, 2010.

[38] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *ACM SOSP*, 2009.

[39] S. Choi, X. Long, M. Shahbaz, S. Booth, A. Keep, J. Marshall, and C. Kim, "The Case for a Flexible Low-Level Backend for Software Data Planes," in *APNET*, 2017.

[40] F. Zane, G. Narlikar, and A. Basu, "CoolCams: Power-efficient TCAMs for Forwarding Engines," in *IEEE INFOCOM*, 2003.

[41] E. Fredkin, "Trie Memory," *Communications of the ACM*, vol. 3, pp. 490–500, 1960.

[42] D. R. Morrison, "PATRICIA – Practical algorithm to retrieve information coded in alphanumeric," *Journal of ACM*, vol. 15, no. 4, pp. 514–534, Oct. 1968.

[43] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," in *ACM SIGCOMM*, 1997.

[44] M. Waldvogel, G. Varghese, J. Turner, and B. Platter, "Scalable High Speed IP Routing Table Lookups," in *Proceedings of ACM SIGCOMM*, 1997.

[45] V. Srinivasan and G. Varghese, "Fast IP Lookups Using Controlled Prefix Expansion," in *Proceedings of ACM SIGMETRICS*, 1998.

[46] S. Nilsson and G. Karlsson, "IP-address Lookup Using LC-tries," *IEEE J.Sel. A. Commun.*, vol. 17, no. 6, pp. 1083–1092, Sep. 2006.

[47] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast Packet Classification Using Bloom Filters," in *ACM/IEEE ANCS*, 2006.

[48] N. Tata, "MicroCuckoo Hash Engine for High-Speed IP Lookup," *MS Thesis at Virginia Tech*, May 2017.