Fast OpenFlow Table Lookup with Fast Update

Tong Yang*† Alex X. Liu‡ Yulong Shen¶ Qiaobin Fu‖ Dagang Li** Xiaoming Li*
*Department of Computer Science, Peking University, China
†Collaborative Innovation Center of High Performance Computing, NUDT, China
‡Department of Computer Science and Engineering, Michigan State University, USA
¶Xidian University, China, ‖Boston University, USA
**Peking University Shenzhen Graduate School, China

Abstract—**Software-Defined Networking (SDN), which separates the control plane and data plane, is a promising new network architecture for the Future Internet. OpenFlow is the *de facto* standard which defines the communication protocol between the controller and switches. The most challenging issue in OpenFlow switches is the lookup of multiple OpenFlow tables. The lookup of OpenFlow tables is so complicated that the state-of-the-art research are still focusing on the design of lookup pipeline architecture, and there is no specific algorithm for the lookup of OpenFlow tables. In this paper, we revise the long-pipeline architecture of OpenFlow 1.4 to a 5-stage pipeline architecture to make a trade-off between flexibility and implementability, and decompose the lookup of OpenFlow tables into three kinds of lookup: longest prefix matching (IP lookup), multi-field matching (packet classification), and exact matching. Then we design new algorithms for packet classification, because the state-of-the-art solutions for them seldom support fast update which is highly demanding for OpenFlow. The other two kinds of lookups can be well handled by state-of-the-art. Experimental results show that our proposed algorithms work excellently, and outperform state-of-the-art solutions.**

I. Introduction

A. Background and Motivation

In recent years, Software-Defined Networking (SDN) is proposed to separate the control plane and data plane of routers. The control planes are centralized and named as controller. The data planes are distributed and named as switch. OpenFlow is the *de facto* standard which defines the communication mechanism and message format between the controllers and the switches.

In OpenFlow switches, the most challenging issue is the lookup of OpenFlow tables. As shown in Figure 1, OpenFlow switch specification 1.4 defines an architecture with many pipeline stages, each of which holds one flow table [1]. One incoming packet could be checked by multiple flow tables. Such a long-pipeline architecture is difficult to implement in practice. The state-of-the-art solution is still designing the lookup architecture [2], and there is no well-known specific lookup algorithm for OpenFlow tables.

To address the problem of OpenFlow tables lookup, two harsh requirements must be satisfied. First, the lookup architecture should be implementable in practice. Second, the lookup algorithms for any kinds of OpenFlow tables should be fast and support fast incremental update.

OpenFlow 1.4 defines the long-pipeline architecture, which is hard to implement in practice, because there could be many different OpenFlow tables in one switch. Among so many OpenFlow tables, the lookup of Forwarding Information Bases (FIBs) and packet classifiers are two most challenging issues. Although there are various solutions, very few algorithms for FIB support fast lookup and fast update at the same time, and almost no algorithms for classifier supports incremental update. In sum, prior art do not satisfy the above two requirements.

B. Proposed Solutions

First, we observe that although there could be many OpenFlow tables, most packets only need to query a few of them. Based on this observation, we propose to divide the tables into several groups, at most one table in each group will be queried. Then we put the tables belonging to the same group in the same pipeline stage. In this way, the long-pipeline architecture is changed into a short-pipeline one at the cost of losing little flexibility. Among various OpenFlow tables, the most difficult problems are to lookup the FIBs and classifiers. For the lookup of FIB (also known as IP lookup), we use the SAIL_L algorithm [3] to perform fast lookup and fast update.

Second, we propose a Bloom filter intersection algorithm for packet classification. We first decompose the lookup of classifiers into per-field lookup, and then use Bloom filter to compute the intersection of all the per-field outputs. The per-field lookup of classifiers can be divided into three categories: Prefix Matching (PM) for the source and destination IP addresses, Range Matching (RM) for the source and destination ports, and Exact Matching for the protocol type. For PM, we use SAIL_L to achieve fast lookup and update. For RM, we propose a scheme named `full range coding` to support fast lookup and propose a scheme named `Common Prefix Matching` (CPC) to support update. EM can be simply solved by hash tables.

*Corresponding author: Yulong Shen (Email: ylshen@mail.xidian.edu.cn). This work is partially supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (973 Program) under Grant No. 2014CB340405, NSFC (61672061, 61472184, 61321491), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, China, NSF (1345307, CNS-1318563, CNS-1524698, CNS-1421407, IIP-1632051), and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. Alex X. Liu is also affiliated with the State Key Laboratory for Novel Software Technology, Nanjing University, China.

Third, there are frequent AND and OR operations of Bloom filters in our scheme. To speed up such operations, we propose to use SIMD (single instruction, multiple data) by taking advantage of CPU pipeline technique. Specifically, we make use of eight 128-bit registers to perform AND and OR operations of Bloom filters in the CPU pipeline.

Paper organization. The rest of the paper proceeds as follows. In section II, we present our short-pipeline architecture for OpenFlow tables. A set of algorithms for packet classification in section III. Section IV shows the implementation details of our algorithms. We present experimental results in section V. Section VI surveys the related work. Finally, we conclude the paper in section VII.

II. Architecture of OpenFlow Tables Lookup

A. OpenFlow Switch Specification 1.4

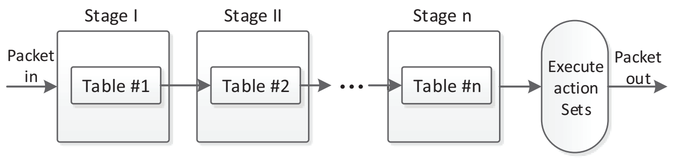


Fig. 1. Pipeline architecture for the lookup of OpenFlow tables.

An OpenFlow switch holds many tables, such as FIB, ACL, MAC address, and VLAN tag tables, etc. As shown in Figure 1, given one incoming packet, one or more OpenFlow tables may need to be queried. The number of alternative fields increases to 42 in OpenFlow Switch Specification 1.4 [1]. One OpenFlow table could contain one or more fields of the 42 fields. OpenFlow Switch Specification 1.4 presents a straightforward solution – assigning one pipeline stage for each table. The pipeline starts from the first table – table #1, whether other tables will be used or not depends on the outcome of the former table. The main disadvantage of this pipeline architecture is that the number of stages could be too many to be implemented in practice. To address this issue, we propose to reduce the number of stages in the following section.

B. Modified OpenFlow Switch Architecture

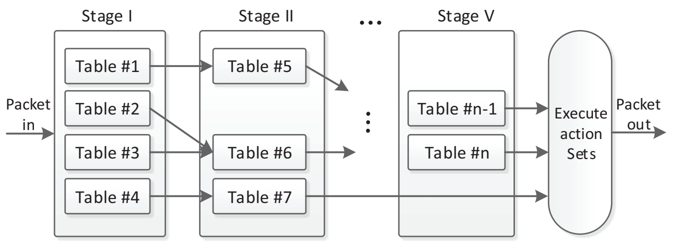


Fig. 2. Modified architecture for the lookup of OpenFlow tables.

In practice, one incoming packet only needs to query a few OpenFlow tables. Therefore, as shown in Figure 2, we can classify the tables into several groups, and make sure that at most one table in a group needs to be checked for one packet. Then we build one pipeline stage for each group. Note that the pipeline processing is only allowed to go forward but not backward. When inserting a new table, we need to decide which stage/group this table should be inserted into. For example, given an incoming packet, it should first check the VLAN tag table in the first stage to decide which classifier in the second stage should be used, and finally, the corresponding FIB in the third stages will be queried to decide the egress port. Generally, five stages are sufficient for most applications. This architecture significantly reduces the number of pipeline stages, but could sacrifice a little flexibility because at most five tables can be queried for a packet. However, it is believed that well-designed 5-stage tables can satisfy the requirements of most application scenarios. In other words, this architecture provides an opportunity to perform efficient lookup and update of OpenFlow tables by sacrificing little flexibility. Current commercial SDN switches support such a pipeline design [4].

C. Decomposing OpenFlow Tables Lookup

The OpenFlow tables can be classified into two categories: single-field tables (*e.g.,* FIBs, MAC tables) and multiple-field tables (*e.g.*, classifiers). The lookup of single-field tables can be divided into two categories: Longest Prefix Matching (*e.g.,* IP Lookup), and Exact Matching (*e.g.,* MAC address table). The problem of Exact Matching can be efficiently solved by hash tables. IP lookup and packet classification are two classic problems, and various solutions have been proposed. For the lookup of OpenFlow tables, the update performance is demanding because the controller could frequently announce rules to OpenFlow switches. Unfortunately, few IP lookup algorithms support fast update, and almost no prior packet classification algorithms supports incremental update. To support fast lookup and update, we use SAIL_L[2] for IP lookup and propose Bloom Filter Intersection (BFI) algorithm for packet classification, and we name our solution Bloom filter Intersection (BFI).

III. Packet Classification

A. Background

Since our scheme for packet classification is based on BV [5] and its successor ABV [6], we first present some background details on these two algorithms.

1) BV scheme: Lakshman and Stiliadis [5] proposed the seminal technique which is commonly referred to as Lucent bitvector scheme (BV). The technique proceeds in two steps: First, given a classifier with n rules, for an incoming 5-tuple tp, BV conducts lookup in every individual field. The lookup result of each field is recorded in a $n-$bit bit vector which is initialized to be all 0s. If tp matches rule r_i, the $i-$th bit of the bit vector is set to 1. Second, BV makes a bitwise logical AND (&) of all the bit vectors. The most significant "1"

[2]The SAIL_L algorithm [3] is a cross-platform algorithm which can achieve fast and constant IP lookup speed and fast incremental update.

bit in the resulting vector corresponds to the highest priority matching rule. The time and space complexity of searching bit vectors is $O(log_2 n)$ and $O(F * n^2)$, respectively, where n is the number of rules and F is the number of fields. Further, BV proposed a method to reduce the length of bit vectors, while ABV [6] proposed a better one. The authors also proposed an optimization for classifiers only with source and destination IP addresses, which we do not discuss due to space limitation.

BV mainly targets at hardware implementation. The authors implemented a five-field version in FPGA platform, which performs one million lookups per second for 512 rules.

2) ABV scheme: The memory usage and lookup speed of BV scheme can be improved if the length of bit vectors can be reduced. Baboescu and Varghese proposed the ABV scheme to aggregate the bit vector [6]. ABV includes two methods. The first method is called recursive aggregation. For example, given a 8-bit vector 0000 0101, it can be aggregated by 01, where "0" means that the first four bits are all "0" bits, and "1" means that there is at least one "1" bit in the second four bits. In this way, the length of the bit vector is reduced to 1/4. To reduce the number of "1" bits in the aggregated bit vector, the authors proposed the second method – filter rearrangement. It rearranges the filters which are order-independent to make "1"s to be relatively intensive, and ultimately reduces the "1"s of the aggregated bit vector. The first method only works when the "1" bits are sparse, but the total memory usage of bit vectors is kept unchanged. The second method comes at a cost of slowing down the update speed.

We propose to use Bloom filters to replace bit vectors, and then make the bitwise `AND` operation of Bloom filters. There is a widely known property that the maximum number of filters matching a packet is inherently limited in real filter sets [7]. Therefore, the number of rule IDs inserted into Bloom filters is much smaller than n, thus the size of Bloom filters is also much smaller than n, especially when n is large. Further, we propose to use SIMD to speed up the logical `AND` and `OR` operations. Compared with ABV, our Bloom filter scheme not only speeds up the query, but also reduces the space complexity from $O(F * n^2)$ to $O(F * m * n)$, where m is the size of Bloom filters.

B. Proposed Architecture of Packet Classification

A classifier $\mathcal{K}$ consists of n rules, denoted by $r_1, ..., r_n$. Each rule consists of F fields and corresponds to an action. Each rule associates with a priority. When multiple rules are matched, the action of the rule with the highest priority is executed. One commonly used method is to let the rule with smaller ID have higher priority.

In this section, we proposed a solution named Bloom Filter Intersection (BFI) [8], [9] for packet classification. Figure 3 shows an example of the architecture of our BFI solution for 5-tuple classifiers. Our solution proceeds in five steps. First, we split the classifier $\mathcal{K}$ with F fields into F per-field tables. We conduct lookup in every per-field table, and report the IDs of matched rules (ID set). Second, we build one Bloom filter (BF) for each ID set. Third, we compute the intersection BF by

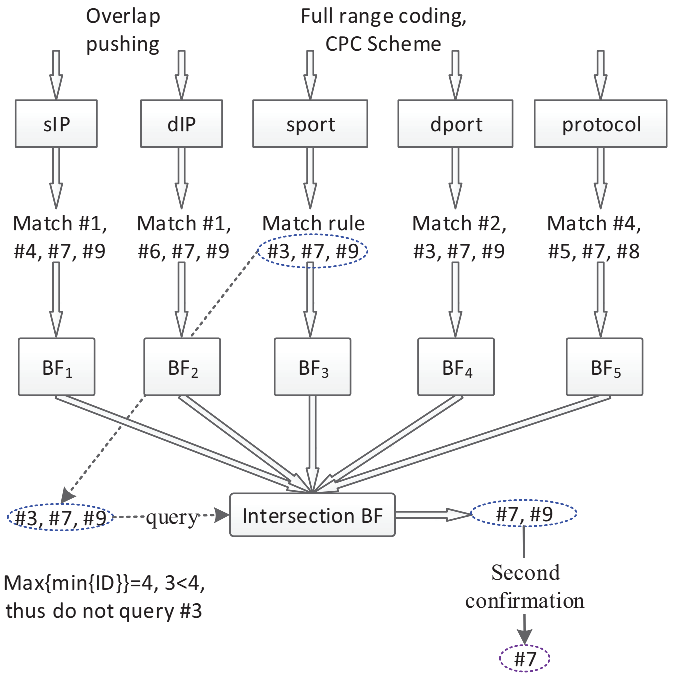


Fig. 3. Proposed architecture for packet classification.

making bitwise `AND` of all BFs. Fourth, we query the smallest ID set in the intersection BF, and figure out the intersection of ID sets. Fifth, we perform a second confirmation by comparing the packet head with the matched rules one by one, and then output the smallest rule ID.

For the lookup of per-field tables, there are three kinds of matching: Prefix Matching (PM), Range Matching (RM), and Exact Matching (EM). Different from Longest Prefix Matching (LPM), PM reports all the matched prefixes. EM is simple and thus we do not discuss it in this paper.

PM or LPM are well-studied problems. The recent works are multi-core solutions [10], FIB compression [11] and SAIL algorithms [3]. We use the SAIL_L algoirthm to address the PM problem. For the RM problem, there are two typical algorithms: binary code [12] and gray code [13], [14]. Unfortunately, both of them cannot support fast incremental update. To address this issue, we propose the common prefix coding scheme, namely CPC, to support fast update.

Next, we present three key schemes of our BFI solution: overlapped pushing, CPC and Intersection BF.

C. Overlap Pushing Scheme

Note that searching the fields of sIP and dIP is a PM problem rather than an LPM problem. Therefore, we propose a pushing method – `Overlap Pushing`. The only difference of overlap pushing and conventional pushing is that, using overlap pushing, those nodes which are pushed to level 16, 24 or 32 will inherit all next hops in the pushing path of the chunk. After overlap pushing, each prefix node at level 16, 24, or 32 has a next hop array, which stores multiple rule IDs, and then we can build a vector and a Bloom filter for those prefix nodes. The lookup and update methods of overlap pushing scheme are similar to that of conventional pushing.

D. Full Range Coding and CPC Scheme

The industry standard of classifiers is that only source and destination ports are represented by ranges [15]. Although the source and destination ports only range from 0 to 65535, it is the main reason of causing an entry explosion for TCAM (Ternary Content Addressable Memory) based solutions. There are three well-know range coding schemes: binary code [12], gray code [13] and RFC code [16]. Binary code and gray code can alleviate the entry explosion to some extent. RFC code first splits the whole range into i small ranges, and then builds one bitmap with n bits for each range, where n is the number of rules. This method is not suitable for TCAM but for software. The main shortcoming is too much memory usage. Unfortunately, all of them cannot support fast incremental update. Adding a large range will cause a large number of entry updates. To conduct fast range lookup and update, we propose a coding scheme named common prefixes coding (CPC in short) algorithm.

1) Full Range Coding: Because both source port and destination port range from 0 to 65535. Thus we propose to use a straightforward solution, namely Full Range Coding for convenience: building a full array $R_g[65536]$ with 65536 elements, each element includes a set of IDs. Given an incoming port i, the matched IDs are stored in $R_g[i]$. For example, $R_g[5] = \#3, 7, 9$ means rules 3, 7 and 9 include the port 5. In this way, we can directly get the address of the matched ID set. Further, We can build a small BF for each ID set.

Full Range Coding can achieve fast lookup speed, but brings difficulties for incremental update. In this section, we propose an algorithm named CPC to minimize the update overhead.

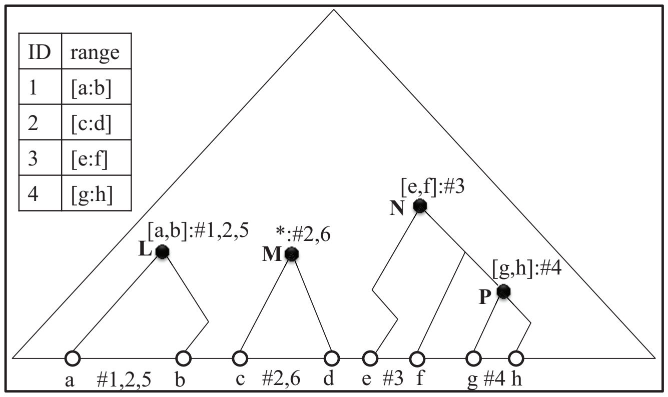


Fig. 4. Coding the Ranges.

2) CPC Scheme: **Definition I**: prefix of a range. Given a range $[a, b]$ ($0 \leqslant a,\ b \leqslant 65535$), we define $pre(a, b)$ as the `longest common binary prefix` of a and b, while a and b are represented by 16-bit binary digitals.

For example, given a range [10011, 10100], then the prefix of this range $pre(10011, 10100)$ is 10.

Theorem 3.1: Given any two ranges: $[a, b]$ and $[c, d]$, if they have no intersection, then their common prefixes are different, *i.e.*, $pre(a, b) \neq pre(c, d)$.

Proof: Assume $pre(a, b) = pre(c, d) = p$, and we assume the length of p is l. The $(l + 1)$-th bit of a and b must be 0 and 1, because p is their longest common binary prefix. Thus a and b can be represented by $p0*$, and $p1*$. Since we assume $pre(a, b) = pre(c, d)$, for the same reason, c and d can also be represented by $p0*$, and $p1*$. In this case, the digital with prefix $p1$ and other bits all 0s – $p1000...$ must be in range $p0* \sim p1*$. In other words, $p1000... \in [a, b]$ and $p1000... \in [c, d]$. This contradicts with the premise of no intersection. Therefore, the assumption $pre(a, b) = pre(c, d)$ does not hold, *i.e.*, $pre(a, b) \neq pre(c, d)$. ■

3) Construction of CPC: As shown in Figure 4, the construction of our CPC scheme proceeds in the following steps. First, we record all the ranges of the classifier, and suppose we get g_0 ranges. Such g_0 ranges divide the whole range [0,65535] into g non-overlapping sub-ranges. Second, we represent all the g ranges by their `common prefixes`, and there will be g prefixes. These prefixes can be represented by a 16-bit trie, and we call it CPC trie for convenience. Specifically, given any small range [a,b]:IDs, we compute $pre(a, b)$, and insert a node corresponding to $pre(a, b)$ into the CPC trie, and set the data field to [a,b]:IDs.

Given a port i, which one of the g ranges contains i? The answer is very interesting, we can lookup i in the CPC trie, and the longest matched prefix represents the matched range. In other words, the initial version of CPC scheme converts range matching into LPM.

Note that the CPC scheme is proposed to only assist the update of Full Range Coding, thus we show the update mechanism as follows.

4) Using CPC trie to Assist Range Updating: When inserting a new range [a,b]:id, one straightforward solution is to update those ranges which have intersections with [a,b]:id. Obviously, updating so many ranges will degrade the update performance. To minimize the update overhead, we propose to first compute $pre(a, b)$, and then only insert the corresponding node into the CPC trie. In this way, the lookup no longer follows the LPM rule but follows the PM rule. The lookup must check all the matched prefixes to judge whether the incoming port is in the ranges stored in the prefix node, and then returns all the matched IDs. When deleting a range [a,b]:id, we do not actually perform deletion, but insert [a,b]:-id instead. When the accessed CPC trie node has a minus ID, this means the ID is already deleted.

Because the ranges are initially coded by our Full Range Coding method, thus the number of prefixes in the CPC trie is equal to the number of insertions. For the lookup of CPC trie, we can just lookup the CPC trie using CPUs, and we can also use FPGA to accelerate the lookup speed, given the CPC trie is usually small enough to be held in the on-chip memory of FPGA. Using FPGA, one lookup of CPC trie can be achieved in one clock cycle when pipelining. The worst case update is 16 memory accesses. When there is an update burst, the update performance can be regarded as $O(1)$ because the update handling is also pipelined during burst. After long-term updating, the size of the CPC trie may be too large to be held in the on-chip memory of FPGA, in this case, we can perform one re-construction using the Full Range Coding scheme to empty the CPC trie.

E. Results Intersection

1) Intersection Bloom Filter: We analyze the problem of results aggregation of all per-field results in this section. This is the same as the problem of sets intersection. Therefore, we state the problem as: given c sets, each set has n_i $(1 \le i \le c)$ elements, what is the intersection of the c sets? We propose to use Bloom Filters [8], [17] (BF in short) to address this problem. And the solution has three versions as follows.

1. &BF. The straightforward solution is to build Bloom Filters for sets, and then AND them, thus we call it &BF. Specifically, given $c-1$ sets: S_1, $S_2...S_{c-1}$, &BF first builds $c-1$ Bloom Filters: BF$_1$, BF$_2$...BF$_{c-1}$ using the same k hash functions and the same amount of memory – m bits. Then &BF makes a bitwise AND operation for all the $c-1$ BFs and obtains the resulting BF, namely BF^{c-1}. Last, it queries BF^{c-1} using all the elements of S_c , and then outputs the elements when BF^{c-1} reports true. This version is simple and fast, but there is room for improvement.

2. Version I – Selected &BF. Compared with &BF, our first version adds an additional selection operation, thus we call it Selected &BF. The Selected &BF scheme proceeds in the following four steps. First, Selected &BF constructs one BF per set. Second, it makes a bitwise AND of all the c BFs, and gets a resulting BF – BF^c. Third, it queries BF^c using the smallest set.

Here we compare the size of intersection result of &BF and Selected &BF. First, FPR comparison. Note that BF^c is the result of BF^{c-1}&BF$_c$, thus there are fewer $1s$ in BF^c. That is to say, the FPR of BF^c is smaller than that of BF^{c-1}. &BF uses BF^{c-1} and Selected &BF uses BF^c, thus Selected &BF has smaller FPR. Second, querying sets comparison. &BF checks all the elements of the last set, where the last set is not always the smallest. In contrast, Selected &BF checks all the elements of the smallest set. In conclusion, Selected &BF outputs a smaller intersection than &BF.

3. Version II – Priority Selected &BF. To further minimize the query overhead, besides choosing the smallest sets, we figure out the largest one of minimal ID in each ID set, we name it $\max_{i=1}^{F}\{\min_{ID\in S_i}\{ID\}\}$. For the smallest ID set, we only check those IDs which are equal to or larger than $\max_{i=1}^{F}\{\min_{ID\in S_i}\{ID\}\}$. Note that we will sort the element (ID) in ascending order during the insertion, and always query the ID from the smallest one.

4. Using Different k for Different Priorities. After the intersection BF reports the matched IDs, we need to make a second confirmation from the rule with highest priority (*i.e.*, smallest ID). If the rules with higher priority have smaller FPR, the number of second confirmations can be reduced. Therefore, during the construction of Bloom Filters, we assign more hash functions for smaller IDs. One simple approach is that: Given an ID id and k_0 independent hash functions, we assign the first $k_0 - \lfloor \log_2 id \rfloor$ hash functions for it.

A similar idea proposed by Li *et al.*. is multi-class Bloom Filter [18], it chooses the number of hash functions k based on the probability of each element to be inserted into the Bloom Filter. In other words, multi-class BF chooses k by the probability that elements appear, while our improved BF chooses k by the priority of elements.

2) The Formula of Intersection of Bloom Filters: The false positive rate of the intersection of Bloom Filters can be computed by the following theorem.

Theorem 1: Given two sets S$_1$ with n_1 elements and S$_2$ with n_2 elements. We build two Bloom Filters (BF$_1$ for S$_1$ and BF$_2$ for S$_2$) using the same k independent hash functions. The size of both BF$_1$ and BF$_2$ is m bits. We make bitwise AND operation of BF$_1$ and BF$_2$, and we name the resulting BF as *r*BF. Let f_1, f_2 and f_r represent the FPR of BF$_1$, BF$_2$ and BF$_r$. It can be concluded that $f_r = f_1 * f_2$.

Proof: We define p' as the probability that one bit (suppose it is at position i) is 0 in *r*BF. There are two cases:

Case I: the bit at position i is zero in both BF$_1$ and BF$_2$. For an arbitrary element e in S$_1$ or S$_2$, if $h_i(e)$ does not point to i, then the bit at position i is still zero. The probability is

$$p'_I = \left(\frac{m-1}{m}\right)^{k(n_1+n_2)} = \left(1-\frac{1}{m}\right)^{k(n_1+n_2)} \tag{1}$$

Case II: the bit at position i is zero in one BF of BF$_1$ and BF$_2$. The probability is

$$\begin{aligned} p'_{II} = &\left(1-\left(\frac{m-1}{m}\right)^{kn_1}\right)\left(\frac{m-1}{m}\right)^{kn_2} + \\ &\left(1-\left(\frac{m-1}{m}\right)^{kn_2}\right)\left(\frac{m-1}{m}\right)^{kn_1} \end{aligned} \tag{2}$$

Obviously, $p' = p'_I + p'_{II}$, therefore, we have

$$\begin{aligned} &1-p' = 1-p'_I-p'_{II} \\ &= 1+\left(\frac{m-1}{m}\right)^{k(n_1+n_2)} - \left(\frac{m-1}{m}\right)^{kn_1} - \left(\frac{m-1}{m}\right)^{kn_2} \\ &= \left(1-\left(1-\frac{1}{m}\right)^{kn_1}\right) * \left(1-\left(1-\frac{1}{m}\right)^{kn_2}\right) \end{aligned} \tag{3}$$

And finally, the false positive rate of the intersection BF is given by the following equation.

$$\begin{aligned} f &= (1-p')^k \\ &= \left(1-\left(1-\frac{1}{m}\right)^{kn_1}\right)^k * \left(1-\left(1-\frac{1}{m}\right)^{kn_2}\right)^k \\ &= f_1 * f_2 \end{aligned} \tag{4}$$

■

Note that the above derivation of FPRs is based on the original Bloom's FPR formula [8], which was claimed to be flawed by Bose *et al.* [19] and Christensen *et al.* [20]. However, both studies pointed out that the error of Bloom's formula is negligible, especially when m is large, thus we still use Bloom's formula in this paper.

3) Second Confirmation after the Report of Bloom Filters: The Bloom filters will report which IDs are matched. If false positives occur, we need to make a second confirmation. The method proceeds in the following three steps. First, we store all rules in advance in the format of bit arrays. Given one 5-tuple rule, we combine sIP, dIP and protocol into a 72-bit array (`Arr72` in short), note that the "*"s are replaced by 1s. At the same time, we constitute the corresponding 72-bit mask (`Mask72` in short). Note that the if the mask is 3, we use 0001111111.... to represent it. Second, given an incoming packet, we combine its sIP, dIP, and protocol into a 72-bit array (`incoming72`), then we compute "`(incoming72 OR mask72) XOR Arr72`", if the result is 0, go to the third step; otherwise, report false. Third, we check whether the source and destination ports of the incoming packet match the rule by simple comparison. If the result is negative, we check the next rule. Otherwise, we report the ID, and algorithm ends.

IV. Implementation

Since our algorithm needs frequent bitmap operations, we propose to use SIMD to accelerate the speed of logical `AND` and `OR` of bitmaps. We also show how to use FPGA to accelerate the update speed of our BFI scheme.

A. using SIMD on CPU Platform

The bottleneck of our BFI algorithm is the `AND` operation of Bloom filters. To address this issue, we propose to use Single Instruction Multiple Data (SIMD) [21] to speed up such operations. The first instance of SIMD extensions to the $\times 86$ architecture was called MMX in 1995. Eight 128-bit registers (XMM0 $\sim$ XMM7) are added to the later SIMD extensions (SSE, SSE2, and SSE3). The SIMD operation enables processing of multiple data with a single instruction, which means that they are faster although they take more clock cycles to execute than a simple $\times 86$ operation.

We can also take advantage of CPU pipeline technique. Operations of modern processors are broken into multiple micro operations (μ-ops). Each single μ-op takes 1 cycle to execute. Suppose we want to do two subsequent `AND` operations:

`pand` A, B;
`pand` C, D;

Each `pand` operation takes t μ-ops. CPU does not wait t μ-ops for the first operation and then work on the second operation. If the `pand` operation needs to wait v cycles(usually much less than t) before handing the next instruction in the pipeline, then two `pand` operations only take $t + v$ cycles.

In our scheme, we need to perform frequent logical `AND` and `OR` operations on Bloom filters. Thus we propose to use SIMD and the CPU pipeline technique to speed up such operations.

TABLE I
The assembly code of bitwise AND of bit vector A and B.

Coding schemes

```
_asm
{
                 mov       edi,            A;
                 mov       esi,            B;
                 mov       ecx,            counter;
                 jmp       ZERO1;
Loop1:
                 movapd    xmm0,           [edi];
                 movapd    xmm1,           [edi + 16];
                 ...
                 movapd    xmm7,           [edi + 112];

                 pand      xmm0,           [esi];
                 pand      xmm1,           [esi + 16];
                 ...
                 pand      xmm7,           [esi + 112];

                 movapd    [edi],          xmm0;
                 movapd    [edi + 16],     xmm1;
                 ...
                 movapd    [edi + 112],    xmm7;
                 add       edi,            128;
                 add       esi,            128;
ZERO1:
                 dec       ecx;
                 jns       Loop1;
}
```

Taking a simple example, given two bit vectors A and B, we show the assembly code of bitwise `AND` of vectors A and B in Table I. The assembly code proceeds in the following steps. First, we copy each 128 bits of A to 8 registers (XMM0 $\sim$ XMM7). Second, we perform `AND` operation in the above registers in parallel. Third, we go to the next loop till all the bits of both A and B are processed. In this way, currently we can make bitwise `AND` for two 128*8-bit vectors almost in one cycle. In other words, this approach extends the size of a machine WORD from 64 bits to 1024 bits. The width of the SIMD registers is increased to 512 bits, and the number is increased to 32 (ZMM0 $\sim$ ZMM31). It is supported by Intel's Knights Landing processor. Thus our scheme can be further improved.

B. Implementation on FPGA Platform

Although our BFI scheme can perform excellently only using CPUs, it provides the opportunity to accelerate the update speed when using FPGA.

During the update process, we build two CPC tries for the fields of sport and dport. Similar to prior SRAM-pipeline scheme for IP lookup, we build one stage per level in the on-chip memory of FPGA. Because we only store the insertion of update messages in the CPC trie, and each CPC trie has only 16 levels. When the input packets are continuous, such pipelined stages works, the system throughput is equal to that of system frequency. We carry out the FPGA simulation for Virtex-7 (model XC7VX1140T) using Xilinx ISE 13.2

IDE., the integration simulation results of 10K rules shows a minimum clock period of 3.79ns, *i.e.*, a maximum frequency of 264 MHz.

V. Experimental Results

Since we decompose the lookup of openflow tables into three kinds of lookups: exact matching (EM), longest prefix matching (LPM), and packet classification. EM can be well handled by hash tables, and we use SAIL_L to deal with LPM, thus we only show the experimental results of packet classification in this section.

A. Experimental Setup

Synthetic classifiers and traffic: To test the performance of our algorithm and prior art, we use the synthetic rule generated by the well-known ClassBench [22]. In our experiments, the rules we have used contain Accesses Control List (ACL), Firewall (FW) and IP Chain (IPC). The size of rule sets ranges from 100 to 10000. Given we can hardly access to real-world classifiers, the seed file which ClassBench provided can make the performance as close to practice as possible.
Computing platform: We carry out our experiments on a desktop computer equipped with a Pentium (R) Dual-Core CPU 5500@2.80GHz and 8GB Memory running Windows 7.
Source codes: The source codes of SAIL_L, Lulea, TreeBitmap, LC-trie were downloaded from [23]. The source code of BV was downloaded from [24], and we implemented ABV by ourselves. We obtain the source code of efficut and hypercut written by the authors.

B. Experiments on Packet Classification

We compare our scheme BFI with the well-known software algorithms: BV [5], ABV [6], hypercuts [25], efficuts [26]. The source code of hypersplit released by the authors [27] does not always output the correct rule ID, thus is not compared with our algorithm. The Hicut [28] takes more than 20 hours to build the decision tree for large rule sets, thus is also not compared with our algorithm. BV and ABV support incremental updates at the cost of huge memory consumption and low lookup speed, while hicuts, hypercuts, efficuts, and hypersplit can achieve relatively high lookup speed at the cost of not supporting incremental update.

To make a fair comparison among these well-known algorithms, we produce different types of rules and the corresponding traffic traces using ClassBench [22], and then test their lookup speed. There is no update trace available and ClassBench cannot produce updates, thus we will evaluate the update performance of packet classification algorithm in the future work.

The experimental results of lookup speed are shown in Figure 5. The x-axis represents the rule set. In this class bench, there are three kinds of rules: ACL rules (ACL), fireware rules (FW), and IPC rules (IPC). We carry out the experiments on these three kinds of rules with the size of 100, 1000, and 10000, respecitively. As shown in Figure 5, for example, ACL100_2 refers to the second sample of ACL rule set with 100 rules, and IPC1000_3 referes to the third sample of IPC rules. We use the corresponding traffic traces to lookup the rule set, and the y-axis represents the lookup speed, where Mpps means Million packets per second. It can be observed that our BFI algorithm outperforms prior art significantly. Specifically, our experimental results show that BFI achieves 10.6∼21.7, 7.4∼8.4, 14.8∼19.4, and 14.1∼24.9 times faster lookup speed compared to BV, ABV, EffiCut, and HyperCut, respectively.

The experimental results of update speed are shown in Figure 5. The x-axis represents the rule set. Results show that the update speed of BFI ranges from 1100 to 1400 updates per second. We do not compare with other packet classification algorithms, because their source codes do not include the update algorithm.

VI. Related Work

We survey the related work from the following three aspects: openflow table lookup, IP lookup, and packet classification.

A. OpenFlow Tables Lookup

For the lookup of OpenFlow table, given an incoming packet, the packet head could be checked by several tables, such as FIB, classifier, MAC tables, *etc.* The lookup of OpenFlow tables is the most challenging issue in OpenFlow switches, and state-of-the-art work that design a SDN-specific switching chip [2] are not finished yet. There is still no well known specific algorithm for the lookup of OpenFlow table.

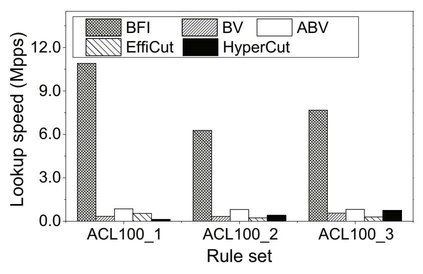
B. IP Lookup

IP lookup is a well-studied problem, and there are various software based and hardware based algorithms. Here we only survey the typical RAM based algorithms. The SIGCOMM'97 best paper proposed Lulea algorithm [29], constructing very small lookup tables using bitmap mechanism, and thus can perform fast lookup due to the cache mechanism of CPUs. We here introduce two deployed algorithms: Tree Bitmap and LC-trie. Tree Bitmap [30] is adopted by Cisco, and LC-trie [31] is adopted in Linux Kernel [32]. Two recent works are the multi-core algorithm [10] and SAIL algorithms [3]. One elaborate survey can be found in [33].

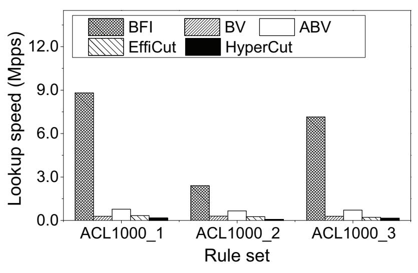
C. Packet Classification

Packet classification algorithms can be classified into two main categories: algorithmic (usually using RAM) solutions and TCAM-based solutions. Two excellent survey papers can be found in [7], [34].

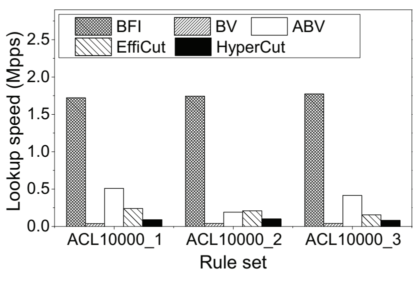
TCAM based solutions are the *de facto* standard in the industrial field. All the rules are stored in TCAM and compared in parallel, the result thereby can be obtained only in one clock cycle [35]–[37]. The shortcomings of TCAM are the high cost and high power consumption, and hard to perform range matching. Binary code [12] and gray code [13] were proposed to enable TCAM support range matching at the cost of entry explosion: one range of W bits is coded to $2W-2$ and $2W-4$ entries in the worst case, respectively. Further improvements proposed by Rottenstreich *et al.* [38], [39] can relieve the entry explosion at the cost of changing the conventional TCAM architecture. The inherent shortcoming of above coding schemes is the slow update performance.



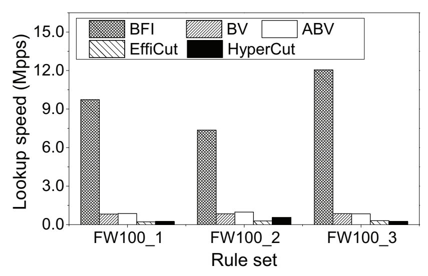
(a) Using 100 ACL rules.



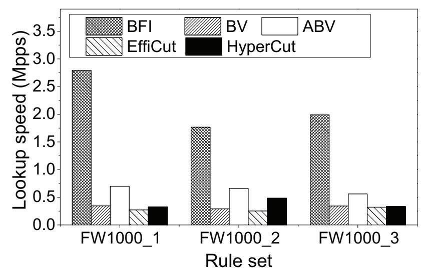
(b) Using 1000 ACL rules.



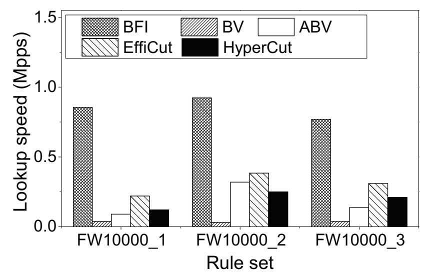
(c) Using 10000 ACL rules.



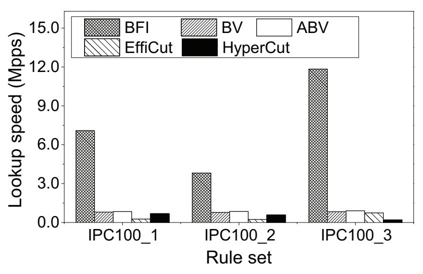
(d) Using 100 firewall rules.



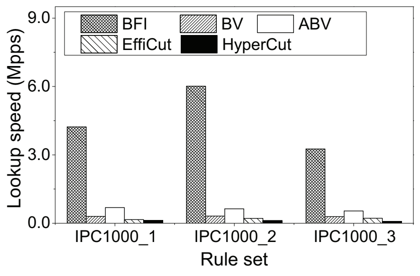
(e) Using 1000 firewall rules.



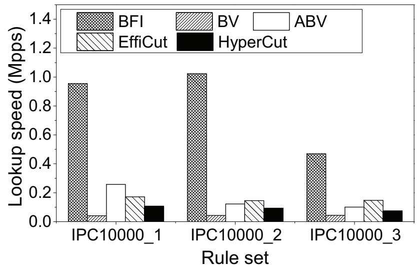
(f) Using 10000 firewall rules.



(g) Using 100 IP chain rules.



(h) Using 1000 IP chain rules.



(i) Using 10000 IP chain rules.

Fig. 5. Lookup speed of packet classification algorithms.

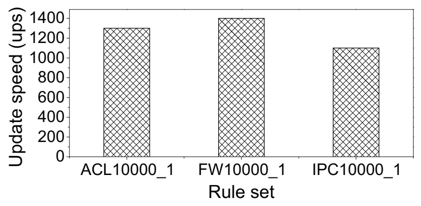


Fig. 6. Update speed of BFI on three different rule sets.

Algorithmic solutions are often implemented in software. Even when the regions of each field are non-overlapping, the best bounds [40] for software solutions are: either $O(n*F)$ space and $O(logn)$ time or $O(n)$ space and $O(log^{F-1}n)$ time, where n is the number of rules and F ($\geq$ 3) is the number of fields. This indicates software solutions suffer either high time complexity or high space complexity. However, Gupta and McKeown introduced a new direction into packet classification research [16] – looking for heuristics that work well on common rule sets. The authors presented the most well known observations regarding the characteristics of real filter sets. After that, most RAM based solutions for faster lookup focus on leveraging the characteristics of real filter sets. Other important observations on real classifiers include [41]–[43]. The observation [15] is that most rules in a classifier are order-independent. In summary, prior software solutions cannot achieve fast search speed when the real classifier does not have the corresponding characteristics. Our BFI algorithm is independent with the characteristics of classifiers.

Other algorithmic solutions include decision trees, bit vectors, and cross-producting. Hicuts [28] is the seminal technique of decision tree based solutions, its successors includes hypercuts [25], hypersplits [44], efficuts [26], etc. These solutions seek for various optimization methods to reduce the memory usage and depth of the decision tree. Peng He *et al.* found that the performance of the above decision tree based solutions varies a lot for the same size classifiers with different characteristics [45]. Bit vector schemes include BV [5] and ABV [6] which are detailed before. Cross-producting [16] scheme first searches per-field tables, and then outputs the results of via the lookup of one or more huge cross-producting tables. Improvement to speed up each table lookup is proposed in [46]. The advantage of the cross-producting schemes is fast lookup speed, while the main shortcomings are the huge memory usage and requirement for re-construction when updating.

VII. Conclusion

The key contributions of this paper lie in the following aspects. First, we modify the long-pipeline architecture of OpenFlow switch specification into a short one, so as to make the lookup of OpenFlow tables implementable. Second, for the packet classification, we propose the BFI algorithm and a set of optimizations (including overlap pushing, full range coding, and CPC) to achieve fast lookup and update. Third, we propose to use SIMD to speed up the `AND` and `OR` operations of Bloom filters. Experimental results show that our proposed solutions significantly outperform state-of-the-art solutions.

References

[1] OpenFlow switch specification 1.4.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proc. ACM SIGCOMM*. ACM, 2013, pp. 99–110.

[3] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee ip lookup performance with fib explosion," in *Proc. ACM SIGCOMM*. ACM, 2014, pp. 39–50.

[4] "Intel ethernet switch fm6000 series," https://www.intel.com/content/www/us/en/ethernet-products/switch-silicon/ethernet-switch-fm6000-series-brief.html.

[5] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 203–214, 1998.

[6] F. Baboescu and G. Varghese, "Scalable packet classification," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4. ACM, 2001, pp. 199–210.

[7] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys (CSUR)*, vol. 37, no. 3, pp. 238–275, 2005.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li, "A shifting bloom filter framework for set queries," *Proceedings of the VLDB Endowment*, vol. 9, no. 5, pp. 408–419, 2016.

[10] Z. Marko, R. Luigi, and M. Miljenko, "Dxr: towards a billion routing lookups per second in software," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 29–36, 2012.

[11] T. Yang, B. Yuan, S. Zhang, T. Zhang, R. Duan, Y. Wang, and B. Liu, "Approaching optimal compression with fast update for large scale routing tables," in *IEEE International Workshop on Quality of Service*, 2012, pp. 1–9.

[12] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 135–146, 1999.

[13] A. Bremler-Barr and D. Hendler, "Space-efficient TCAM-based classification using gray coding," *Computers, IEEE Transactions on*, vol. 61, no. 1, pp. 18–30, 2012.

[14] Z. Chen, A. Liu, Z. Li, Y.-j. Choi, H. Sekiya, and L. Jie., "Energy-efficient broadcasting scheme for smart industrial wireless sensor networks," *Mobile Information Systems,2017,(2017-01-23)*, no. 12, pp. 1–17, 2017.

[15] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Sax-pac (scalable and expressive packet classification)," in *Proc. ACM SIGCOMM*, 2014, pp. 15–26.

[16] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, 1999, pp. 147–160.

[17] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li, "A shifting framework for set queries," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–16, 2017.

[18] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang, "Scalable data center multicast using multi-class bloom filter," in *Network Protocols (ICNP), 2011 19th IEEE International Conference on*. IEEE, 2011, pp. 266–275.

[19] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, "On the false-positive rate of bloom filters," *Information Processing Letters*, vol. 108, no. 4, pp. 210–213, 2008.

[20] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a bloom filter," *Information Processing Letters*, vol. 110, no. 21, pp. 944–949, 2010.

[21] SIMD. http://en.wikipedia.org/wiki/SIMD.

[22] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *INFOCOM 2005*, vol. 3. IEEE, 2005, pp. 2068–2079.

[23] "SAIL webpage," http://fi.ict.ac.cn/firg.php?n=PublicationsAmpTalks.OpenSource.

[24] "Evaluation of packet classification algorithms," http://www.arl.wustl.edu/~hs1/PClassEval.html#4._Algorithms.

[25] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003, pp. 213–224.

[26] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 207–218, 2011.

[27] "Code of hypersplit," http://security.riit.tsinghua.edu.cn/share/index.html.

[28] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *Micro, IEEE*, vol. 20, no. 1, pp. 34–41, 2000.

[29] D. Mikael, B. Andrej, C. Svante, and P. Stephen, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, 1997, pp. 3–14.

[30] E. Will, V. George, and D. Zubin, "Tree bitmap: hardware/software IP lookups with incremental updates," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 97–122, 2004.

[31] N. Stefan and K. Gunnar, "IP-address lookup using lc-tries," *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 6, pp. 1083–1092, 1999.

[32] LC-trie in linux kernel. http://lxr.linux.no/linux+v3.13.5/Documentation/networking/fib_trie.txt.

[33] R. Miguel, B. Ernst, and D. Walid, "Survey and taxonomy of IP address lookup algorithms," *Network, IEEE*, vol. 15, no. 2, 2001.

[34] P. Gupta and N. McKeown, "Algorithms for packet classification," *Network, IEEE*, vol. 15, no. 2, pp. 24–32, 2001.

[35] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4, pp. 193–204, 2005.

[36] C. R. Meiners, A. X. Liu, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," in *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*. IEEE, 2007, pp. 266–275.

[37] Y. Ma and S. Banerjee, "A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification," in *Proc. ACM SIGCOMM*. ACM, 2012, pp. 335–346.

[38] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *Computers, IEEE Transactions on*, vol. 62, no. 6, pp. 1127–1140, 2013.

[39] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "On finding an optimal TCAM encoding scheme for packet classification," in *Proc. IEEE INFOCOM*. IEEE, 2013, pp. 2049–2057.

[40] M. H. Overmars and F. A. van der Stappen, "Range searching and point location among fat objects," *Journal of Algorithms*, vol. 21, no. 3, pp. 629–656, 1996.

[41] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to cams?" in *Proc. IEEE INFOCOM*, vol. 1, 2003, pp. 53–63.

[42] J. Van Lunteren and T. Engbersen, "Fast and scalable packet classification," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 4, pp. 560–571, 2003.

[43] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *Proc. IEEE INFOCOM*, vol. 1. IEEE, 2005, pp. 269–280.

[44] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 648–656.

[45] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. IEEE ICNP*, 2014.

[46] J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar, "Chisel: A storage-efficient, collision-free hash-based network processing architecture," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 203–215, 2006.