Computer Networks 92 (2015) 72–88

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Practical flow table aggregation in SDN☆

Shouxi Luo, Hongfang Yu*, Lemin Li

Key Laboratory of Optical Fiber Sensing and Communications, Ministry of Education, University of Electronic Science and Technology of China, Chengdu 611731, PR China

ARTICLE INFO

Article history:
Received 5 February 2015
Revised 8 July 2015
Accepted 18 September 2015
Available online 25 September 2015

Keywords:
SDN
Flow table aggregation
TCAM
Update

ABSTRACT

In OpenFlow-driven SDN, flow tables are TCAM-hungry; commodity switches suffer from limited concrete flow table size. One method for coping with the limitations is to use aggregation schemes to reduce the number of flow entries required to express the same forwarding semantics. Unfortunately, the aggregation of rules would retard table updates and lengthen the updating duration, during which, the data plane is inconsistent with the control plane. Forwarding errors such as *Reachability Failures*, *Forwarding Loops*, *Traffic Isolation* and *Leakage* are prone to occur. Since network updates take place frequently in practice, the aggregation scheme must be efficient and effective.

In this paper, we proposed FFTA (Fast Flow Table Aggregation) and its online companion, iFFTA (incremental FFTA), to make practical flow table aggregation. FFTA is an offline solution performing snapshot aggregation of non-prefix rules by (1) splitting them into *prefix-permutable* partitions in an aggregation-aware manner, and (2) applying optimal prefix-based aggregation techniques, respectively. When some original rules are updated, iFFTA is triggered to incorporate the update immediately by leveraging the *order-independence* relationship and structure information of rules. To the best of our knowledge, iFFTA is the first online aggregation scheme for non-prefix rules. We employed public available prefix rules as well as synthetic non-prefix rules generated with real parameters to evaluate their performances. Extensive experiments demonstrated that FFTA significantly outperforms prior art on both efficiency and effectiveness, while iFFTA greatly simplifies the update of aggregated rules with an acceptable loss of compression ratio. Accordingly, users could make a combination use of FFTA and iFFTA in practice: call iFFTA usually and recall FFTA once the switch is running out of concrete flow table space.

1. Introduction

In OpenFlow-driven SDN, forwarding tables (i.e., flow tables) are TCAM-hungry (Ternary Content-Addressable Memories) since much more header fields are included into the matching fields. For example, there are 12 fields with more than 237 bits in the first stable version of OpenFlow (i.e., 1.0.0 [1]), and the fields continue to grow as more fields are added in [2,3]. Unfortunately, because TCAMs are

☆ The preliminary version of this paper titled "Fast Incremental Flow Table Aggregation in SDN" was published in the proceedings of the 23rd ICCCN, 2014. In this extended version, we add the following work. (1) We present more design rationales about how to split non-prefix rules into *prefix-permutable* partitions. (2) We analyze the aggregation chances between *prefix-permutable* rules, based which, we further design an algorithm that makes *prefix-permutable* rules in an aggregation-aware manner. (3) We design algorithms to simplify the operation of incorporating incremental updates to aggregated tables by using the order-independence between rules. (4) We theoretically analyze the computation complexity of all proposed algorithms and add detailed evaluations for the added algorithms. (5) We discuss the implementation related problems.

* Corresponding author. Tel.: +86 2861830256.

E-mail addresses: rithmns@gmail.com (S. Luo), yuhf2004@gmail.com, yuhf@uestc.edu.cn (H. Yu), lml@uestc.edu.cn (L. Li).

http://dx.doi.org/10.1016/j.comnet.2015.09.016
1389-1286/

board-space costly, power-hungry, and expensive [4–7], commodity OpenFlow switches suffer from restricted concrete flow table space [7–9].

One promising direction in reducing the demands of TCAMs is to use *flow table aggregation*, a technique that merges multiple flow entries into one without modifying the forwarding semantics. Flow table aggregation is a software solution and does not require any change to the OpenFlow protocol, nor to OpenFlow switches. Controllers are easy to implement the aggregation as an optional service.

While a number of literatures have proposed aggregation schemes for traditional prefix IP routing tables [10–12] or non-prefix TCAMs rules or ACLs [4–6,13], the aggregation of flow table in SDN has its own particularities.

1. Firstly, the match fields of flow table are non-prefix since multiple types of field are included (both prefix- and exact- fields). Hence those specified prefix aggregation schemes cannot cope with the demand (e.g., [5,10–12]).
2. Secondly, the potential actions of a flow table are more varied than that of ACLs, which have about 2 or 4 actions in general. So those 2- or 4- action dedicated aggregation schemes do not work well (e.g., [4,13]).
3. Thirdly and crucially, flow table aggregation in SDN is extremely efficiency-critical. This is because forwarding rules are volatile, while the aggregation of flow table will retard table updates and lengthen the updating duration. During the update period, the data plane is inconsistent with the control plane; forwarding errors such as *Reachability Failures*, *Forwarding Loops*, *Traffic Isolation* and *Leakage* are prone to occur [14]. As a result, inefficient offline non-prefix aggregations are inapplicable (e.g., [4,6]).

To achieve practical flow table aggregations, we present a pair of aggregation schemes named FFTA (Fast Flow Table Aggregation) and iFFTA (incremental FFTA) in this paper.

FFTA is an offline aggregation scheme sharing the same high-level idea with the state-of-the-art non-prefix aggregation scheme—bit weaving [6]. Both bit weaving and FFTA make snapshot aggregations by (1) grouping non-prefix rules into *prefix-permutable* partitions, and (2) aggregating each *prefix-permutable* partition with prefix-based techniques. However, FFTA makes two major improvements. Mainly, on the aggregation of each *prefix-permutable* partition, FFTA adopts a tree-based technique derived from ORTC [10] instead of the dynamic programming algorithm [6]. This not only significantly simplifies the computational complexity of aggregation, but also makes the aggregated rules well-structured and easy to update, without any loss of aggregation effectiveness. On the other hand, FFTA splits *prefix-permutable* partitions in an aggregation-aware manner. This greatly increase the chance of successfully aggregations.

As the complement of FFTA, iFFTA is designed to efficiently incorporate incremental updates to the aggregated table. Once original rules are updated, iFFTA employs the order-independence of non-prefix rules [15] to reduce the number of affected partitions, and uses the tree-structural information of aggregated rules to simplify the computation of incorporating the update to its involved partition.

We use prefix rules from Stanford University Backbone Network [16] to test the effectiveness and efficiency of FFTA and iFFTA on aggregating *prefix-permutable* partitions, and use synthetic non-prefix rules from ClassBench [17] (with real parameters) to evaluate their performances on aggregating entire flow tables. Experimental results demonstrate that:

1. On aggregating *prefix-permutable* partitions, FFTA is about 200 × faster than bit weaving, while using much less memories and sharing the same effectiveness.
2. On aggregating entire flow tables, FFTA significantly outperforms bit weaving on the average compression ratio with the improvement ratio up to 48%.
3. On incorporating updates into aggregated tables, with iFFTA, more than half of insertions can be directly incorporated without any modification of the aggregated rules, and each of all other updates can be incorporated by only recomputing one aggregated partition.
4. Furthermore, on incorporating an update to its aggregated partition, iFFTA is about 3 × faster than (recalling) FFTA with an acceptable loss of compression ratio.

Obviously, FFTA is effective for making snapshot aggregation and iFFTA is friendlier for incorporating updates. In practice, incremental updates will cause the aggregated table to drift away from the "optimal" one obtained by FFTA. So, users could make a combination use of FFTA and iFFTA. A simple but feasible design is calling iFFTA usually and recalling FFTA once the switch is running out of concrete flow table space.

The remainder of this paper is organized as follows. We start by giving an overview of the flow table aggregation problem and its motivation examples in Section 2. Then we present the high-level idea of our aggregation schemes in Section 3. After that, we describe the design details of FFTA and iFFTA in Sections 4 and 5, and analyze their worst-case complexity in Section 6. Before evaluating the effectiveness and efficiency of FFTA and iFFTA in Section 8, we also give a brief discussion of the implementation problem in Section 7. Finally, Sections 9 and 10 present related work and conclusions, respectively.

2. Background and motivation

In this section, we start by briefly reviewing the formal model of flow table in Section 2.1, then introduce its aggregation problem in Section 2.2, and finally show the basic idea of how the non-prefix flow table can be aggregated through a motivation example in Section 2.3.

2.1. SDN and flow table

In OpenFlow-driven SDN, forwarding policies are translated into flow tables to act out [18,19]. Roughly, a flow table is a group of prioritized entries, in which each entry can be simplified as a triple tuple ($\langle m, a, z\rangle$) consisting of the match fields (m), specified action (a), and priority (z). In this paper, we call such an entry a *rule*; we also use *rule* to denote the match field (m) of that entry when no ambiguity exists. Formally, a flow table with n rules can be formalized as a sequence of tuples in nonincreasing order of

z	m	a
1	0111	Fwd 1
2	1111	Fwd 1
3	*101	Fwd 2
4	*011	Fwd 1
5	1*0*	Fwd 3
6	1*1*	Fwd 3
7	****	Drop

(a) the original table

z	m	a
1	*101	Fwd 2
2	**11	Fwd 1
3	1***	Fwd 3
4	****	Drop

(b) the aggregated table

Fig. 1. Two semantic equivalent toy flow tables: the left table has more redundancies than the right. In the left table, rules {1, 2, 3, 4} can be permuted into prefixes by swapping the bit position 1 with 4, and rules {5, 6, 7} can be permuted into prefixes by swapping the bit position 2 with 3. So bit weaving would cut it into two *prefix-permutable* partitions—{1, 2, 3, 4} and {5, 6, 7)}.

rule priorities: $\langle m_1, a_1, z_1\rangle, \ldots, \langle m_n, a_n, z_n\rangle$, where $z_1 \leq \cdots \leq z_n$ (with smaller numbers meaning the higher priority), and rules with the same priority value have disjointed match fields. Then how a packet will be handled is exactly defined by the action of the first matched rule.

In current OpenFlow, an action is a sub-collection of instructions supported by switches, e.g., *forwarding*, *drop*, *modification*, *encapsulation*, and *tunnel to controller*. And the match field is a fixed-length and fat ternary string (consisting of 0, 1 and *wildcard* $*$), specifying the ingress port, packet headers (e.g., VLAN ID, Ethernet src/dst addr, 5-tuple etc.), and optionally metadata written by a previous table. To perform fast entry lookup, match fields are suggested to be implemented using TCAMs in practice. Unfortunately, since TCAMs are board-space costly, power-hungry, and expensive [4–7], commodity hardware switches general have limited TCAMs space [7–9].

2.2. The aggregation problem

In today's SDN, the forwarding rules might be generated by different applications, or installed for different destinations or tenants [20–23]. As a result, in some switches, multiple rules might share the same action (e.g., the same next-hop), or overlap on their match fields. In such a switch, a lot of redundancies would exist among the flow table. There is a chance of aggregating rules to reduce the flow table size. For example, if a switch holds a toy flow table with 7 rules as Fig. 1(a) shows, we can replace it with another semantic-equivalent table which only involves 4 rules as Fig. 1(b) shows. In this instance, we reduce the flow table size about 42.9% without changing the forwarding semantics. Moreover, such a scheme does not require a change to the OpenFlow protocol, nor to OpenFlow devices. It is easy to be implemented as an optional service on controllers. We argue that flow table aggregation is a promising direction in reducing the demands of TCAMs.

However, the aggregation of flow table is a hard problem because rules in the table are non-prefix [1,19]. The work by D. A. Applegate et al. has proven that finding the minimized expression for a non-prefix table is NP-hard, even if there are only two types of action [13]. What's worse, since the configurations/rules of practical networks are volatile, how to efficiently incorporate continuous updates into the aggregated table is another hard problem.

2.3. Motivation examples

Previous research addressed the computational challenges of non-prefix aggregation by designing heuristics based on the characteristics of the aggregated rules [4–6,13]. As far as we know, the best practice at present is to employ the prefix-patterns/prefix-information among non-prefix rules to aggregate. Several literatures (e.g., [5,6]) follow this idea and bit weaving [6] is the best reported scheme.

Bit weaving is based on a key observation that, by permuting some of the bit positions, a group of non-prefix rules could be transformed into prefix format simultaneously (i.e., they are *prefix-permutable*). Take the toy non-prefix rules shown in Fig. 1(a) as an example. We can permute $\{r_1, r_2, r_3, r_4\}$ into prefix format by swapping the bit position 1 with 4, and permute $\{r_5, r_6, r_7\}$ into prefix format by swapping the bit position 2 with 3. Motivated by this, bit weaving performs non-prefix aggregations by cutting the non-prefix table into a series of *prefix-permutable* partitions, and then aggregating, respectively. For instance, bit weaving would split the previous table shown in Fig. 1(b) into $\{r_1, r_2, r_3, r_4\}$ and $\{r_5, r_6, r_7\}$, and then employ *bit swapping*, *weighted one-dimensional prefix list minimization algorithm*, and *bit merging* to aggregate.

Our solution shares the same high-level idea with bit weaving. They both split non-prefix rules into *prefix-permutable* partitions to aggregate. However, our method employs a quite different core method on making *prefix-permutable* partitions and on aggregating them. As we will show, our solution is more efficient and effective. Moreover, our aggregation scheme makes the aggregated table well-structured and easy to update. Based on this, we further propose efficient heuristics to handle incremental updates.

3. Design overview

This section firstly introduces the definition and observation of "*prefix-permutable*" in Section 3.1, then presents the framework of our aggregation algorithms in Section 3.2.

3.1. About prefix-permutable

Definition 1 (Prefix-permutable). A group of ternary strings are *prefix-permutable* if there exists a bit position permutation scheme that is able to permute them into prefix format simultaneously.

Denote $x[i]$ to be the i-th ternary bit in ternary string x and $W(x)$ to be the set of bit positions in x whose values are $*$s, i.e. $W(x) \equiv \{i | x[i] = *\}$. Then the observation of checking whether two ternary strings are *prefix-permutable* or not, can be described as follows:

Observation 1. Given ternary string x and y, they are prefix-permutable if and only if $W(x) \subseteq W(y) \vee W(y) \subseteq W(x)$.

Obviously, if $W(x) \subseteq W(y) \vee W(y) \subseteq W(x)$, x and y can be permuted into prefix format by sorting their bit positions in increasing order by the number of ternary strings that have a * in that bit position. Consider $\{*1*1, 01*0\}$ as an example, the numbers of value-* ternary strings on bit position [1,2,3,4] are [1,0,2,0], respectively; then we can transform them into

prefixes–{11∗∗, 100∗} by permuting their bit positions in order [2,4,1,3]. This observation was first reported in [6], and the authors further extended it to a group of ternary strings and proved the following theorem.

Theorem 1. *Given a group G of ternary strings, they are prefix-permutable if and only if* $W(x) \subseteq W(y) \vee W(y) \subseteq W(x)$ *for* $\forall(x \in G, y \in G)$.

Theorem 1 provides a simple way to check whether a group of non-prefixes are *prefix-permutable* or not. As the test of $W(x) \subseteq W(y) \vee W(y) \subseteq W(x)$ can be performed in constant time using bitmap representations of sets, the worst-case time complexity of the test of n non-prefix rules is $O(n^2)$.

By ordering G's non-prefixes in non-decreasing order of their wildcard numbers, we further get this corollary:

Corollary 1. *Given a list* $l = \langle t_1, t_2, \ldots, t_n \rangle$ *of n ternary strings, where* $|W(t_1)| \leq |W(t_2)| \leq \cdots |W(t_n)|$, *they are prefix-permutable if and only if* $W(t_1) \subseteq W(t_2) \subseteq \cdots W(t_n)$.

The corollary indicates that, if a group of rules are *prefix-permutable*, their sets of value-* positions must be in a chain of subset relationships. This provides a more efficient way to check whether rules are *prefix-permutable* or not. For example, by using the binary-search scheme, the worst-case time complexity of the test of $l \cup t_{n+1}$ is $\log(n)$, where $l = \langle t_1, \ldots, t_n \rangle$ is *prefix-permutable* and ordered. Similarly, the worst-case time complexity of the test of n rules would not exceed $n\log(n)$.

3.2. Framework of our solution

Inspired by the above observations, we can split a flow table into *prefix-permutable* partitions, then adopt prefix-based techniques to aggregate. When the original flow table is updated, we can locate the involved partition(s) and revise some aggregated rules to incorporate the update. Correspondingly, our aggregation scheme is made of two modules: FFTA–Fast Flow Table Aggregation, and iFFTA– incremental FFTA.

FFTA is designed to make snapshot aggregation of a table, while iFFTA is designed to handle incremental table updates. Once rules in the original table are inserted, deleted, or modified, iFFTA is triggered to incorporate the update into the aggregated table. To reduce the delay caused by aggregation, iFFTA simplifies both the computation and operation of update incorporation by leveraging the order-independence and structural information of the rules. However, such a method might miss some aggregation chances. Incremental updates would make the aggregated table drift away from the "optimal" one achieved by rerunning FFTA. Fortunately, there is no need to keep a flow table always being best-aggregated in practice. Accordingly, FFTA and iFFTA can cooperate on aggregation. A simple but practicable design is as Fig. 2 shows. When a switch startups, if table aggregation is needed, its flow table is initially aggregated by FFTA; subsequently, incremental updates trigger iFFTA to make efficient incorporations, which might let the aggregated table drift from optimal; once the switch is running out of concrete flow table space, e.g., the number of available table entries FT_{rem} is smaller than a pre-defined threshold $FT_{threshold}$, FFTA is called to make an effective snapshot aggregation.

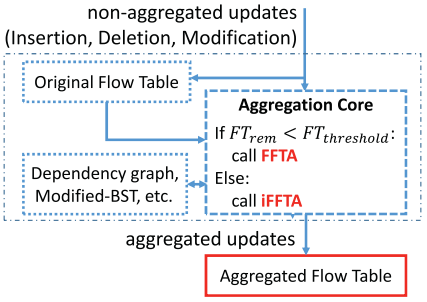


Fig. 2. The framework of how FFTA and iFFTA cooperate.

All relevant technical details of how FFTA and iFFTA work will be discussed in Sections 4 and 5, respectively.

4. Fast snapshot aggregation with offline-FFTA

In this section, we describe the design of FFTA in detail. We firstly introduce how FFTA aggregates a *prefix-permutable* partition in Section 4.1. Then, in Section 4.2, we analyze the aggregation chances between *prefix-permutable* rules and present how FFTA splits a flow table into *prefix-permutable* partitions in an aggregation-aware manner.

4.1. The aggregation of a partition

Key idea. As the non-prefix rules in a partition are *prefix-permutable*, the intuition is to use the best reported prefix snapshot aggregation scheme, ORTC (Optimal Routing Table Constructor) [10], for help. Given a *complete*[1] prefix table, ORTC can aggregate it into an equivalent table that provably contains the minimal number of prefix rules. However, to handle *prefix-permutable* partitions, ORTC meets two challenges. Firstly, ORTC requires the to-be-aggregated rules to be *complete* [10], whereas a partition here is usually incomplete [24], e.g., the one Fig. 4(a) shows. Secondly, ORTC only works on prefix rules, while rules in a partition are generally in non-prefix formats. In rough, FFTA solves the former issue by splitting the *incomplete* partition into the minimal sets of *complete* sub-partitions, and solves the later issue by constructing *prefix-permutable* rules as a modified binary search tree, with which ORTC can deal directly. In the following, we describe them in greater detail.

We start by giving a sketch of ORTC, which drives the design of FFTA. For a given complete prefix table, ORTC builds its rules as a binary search tree (BST), and uses leaf-pushing and relabel techniques to aggregate. As an example, consider the toy prefix table shown in Fig. 3(a). It is complete and its BST is as Fig. 3(b) shows. ORTC performs three passes over the BST to aggregate as Fig. 3 and the below items show:

1. **Pass-1:** Push the nexthop label (i.e. action) from the parents towards the children to expand the prefixes, such that every node in the binary tree either has two or no children.

[1] In [10], a prefix table is called *complete*, iff any packet belonging to the header space of its corresponding BST has a specified action. Similarly, we call a *prefix-permutable* partition *complete* if its permuted prefix rules are *complete*.

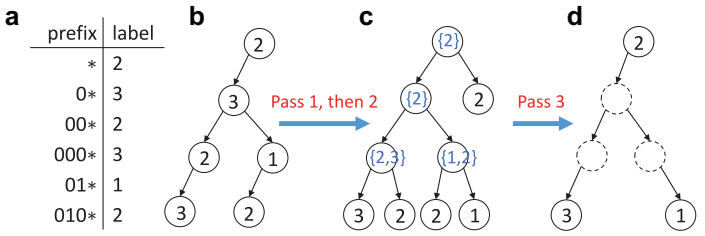


Fig. 3. An example of how ORTC works: (a) tabular form with prefix IP address in binary format and next-hop address label; (b) BST(binary search tree) with state transitions marked; (c) Pass-1 produces the leaf-pushed BST, then Pass-2 gets the set of candidate nexthops for each inner node; (d) and the ORTC-compressed BST.

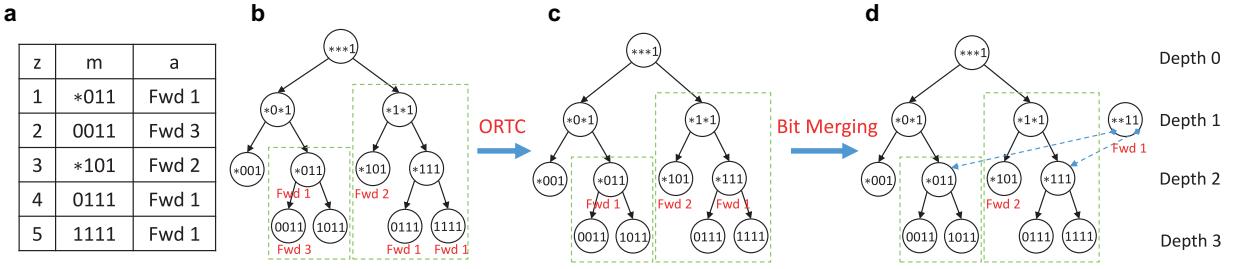


Fig. 4. An example of how FFTA aggregates a prefix-permutable partition: (a) tabular form with non-prefix match field in binary format and action; (b) Modified-BST with action marked; (c) Modified-BST with Modified-ORTC produced; (d) the trace of bit merging and the aggregated Modified-BST.

2. **Pass-2:** Employ a post-order traversal up the tree to get the set of candidate nexthops for each node.
3. **Pass-3:** Assign nexthop to each prefix node in the tree starting from the root and traversing through to the leaves, remove any unnecessary nodes and leaves.

Intuitively, FFTA could built each *prefix-permutable* partition as a BST-alike tree, then run ORTC, respectively. To drive ORTC, the to-be-aggregated rules (i.e., the tree) must be *complete* [10], say, for any packet p belonging to the tree, there is at least one rule in the table that p matches. Unfortunately, *prefix-permutable* partitions are usually incomplete. Take the toy partition shown in Fig. 4(a) as an example. We can build it as a tree as Fig. 4(b) shows. It is obvious that there does not exist any matched rules for packets with header 0001 and 1001. So, ORTC can not work on *prefix-permutable* partitions directly. To deal with this, FFTA first splits the built tree into the minimal list of complete subtrees, where each subtree is complete and contains the maximal number of original rules. We call such a subtree as a maximal complete subtree (MCS). Then FFTA performs ORTC-like techniques on each MCS. After that, FFTA employs a technique named *bit merging* [6] on the aggregated rules to further compress the partition.

In a nutshell, FFTA aggregate a *prefix-permutable* partition by (1) constructing rules as a tree, (2) aggregating the tree with a modification of ORTC, and 3) further compressing these aggregated rules with *bit merging*. Fig. 4 shows a complete example and the operations of each step follow.

Step-1, tree construction. Recall that (1) rules in the partition are able to be permuted into prefixes and (2) these artifactitious prefixes can be organized as a BST, whose each node represents an artifactitious prefix. Denote the match fields of n rules in the partition to be $m_1, m_2, \ldots, m_n$, respectively. Then, the preimage of the lowest common ancestor (LCA) of these artifactitious prefixes in that BST, must be m, where $m = \sum_{i=1}^{n} m_i$. The $+$ operation of ternary string here is defined as follows: for fixed length ternary string x and y, $x + y$ produces a new ternary string z, whose kth bit $z[k]$ is * if $x[k] \neq y[k]$, or $x[k]$ otherwise.

WLOG, we suppose that rules in the partition satisfy $W(m_1) \subseteq W(m_2) \subseteq \cdots W(m_n)$. Obviously, by expanding those *s at bit position $W(m) \backslash W(m_i)$ to 0 and 1, we can build a BST (called Modified-BST) for this partition. Take the toy partition shown in Fig. 4(a) as an example, its LCA is $^{***}1$ and its constructed Modified-BST is shown in Fig. 4(b).

Fig. 5 shows the pseudocode of how FFTA constructs the Modified-BST. In the codes, NODE($rule$, $left = right = nil$) denotes creating a node to store *rule*, and simultaneously letting both its left and right children (denoted as *left* and *right* resp.) be NULL(*nil*). Accordingly, given a node n, $n.m$, $n.a$, and $n.z$ denote the match field(m), action(a), and priority(z) of the rule stored in n, respectively.

To start the procedure of construction, FFTA first calculates the LCA's match field, denoted as m (Line 2), then uses it to create the root node. Initially, the root stores the fake rule $\langle m, nil, \infty \rangle$ (Line 3). After that, FFTA pushes forward the construction by successively appending rules into the tree in nonincreasing order of their amounts of *s. When appending a rule (see APPEND-RULE), FFTA first finds its corresponding node (Line 10), then updates the node's information if the appended rule has the higher priority (Line 11–13). The node finding is performed in a recursive manner (Line 22–26). During the finding, once a node does not exist, FFTA will expand the last visited leaf node and create new nodes (Line 16–21).

Step-2, ORTC-based aggregation. After the Modified-BST is built, FFTA needs to find out the minimal list of MCSs to aggregate. Recall that the Pass-1 of ORTC is to push actions to leaf nodes. According to the definition of completeness and

```
1: function CONSTRUCT-TREE(P)                    ▷ P is a list of rules.
2:     m ← ⋁_{rule∈P} rule.m;                    ▷ calculate the LCA.
3:     root ←NODE(⟨m, nil, ∞⟩, left=right=nil);
4:     for each rule, in nonincreasing of the amount of ∗s do
5:         APPEND-RULE(root, rule);
6:     end for
7:     return root;
8: end function
9: procedure APPEND-RULE(node, rule)
10:    if rule.m = node.m then
11:        if rule.z < node.z then
12:            node.a ← rule.a; node.z ← rule.z;
13:        end if
14:        return
15:    end if
16:    if node.left = nil and node.right = nil then
17:        t ← min{k | node.m[k] ≠ rule.m[k]};
18:        Expand node.m[t] into {0, 1}, get {m^0, m^1} resp.;
19:        node.left ← NODE(⟨m^0, nil, ∞⟩, left=right=nil);
20:        node.right ← NODE(⟨m^1, nil, ∞⟩, left=right=nil);
21:    end if
22:    if rule.m belongs to node.left.m then
23:        APPEND-RULE(node.left, rule);
24:    else
25:        APPEND-RULE(node.right, rule);
26:    end if
27: end procedure
```

Fig. 5. The procedure of constructing the Modified-BST for partition *P*.

```
Modified Pass-1
for each node n (root to leaves, root excluded) do
    p ← parent(n);                    ▷ p is the parent node of n.
    if p.z < n.z then
        n.a ← p.a; n.z ← p.z;
    end if
end for
```

Fig. 6. The pseudo-code of modified Pass-1 of ORTC.

MCS, it is easy to figure out the roots of all MCSs on such a pushed Modified-BST though a post-order traversal. As well, it is obvious that these MCSs must disjoint and their union is just equivalent to the original Modified-BST. Indeed, they are the minimal list of MCSs. Therefore, FFTA firstly pushes actions with Pass-1, then finds all MCSs, and finally runs Pass-2 and Pass-3 on each MCS, respectively.

It should be noted that the Pass-1 employed here is a little different from the one described in [10]. There are two reasons. First, in our Modified-BST, each node is either a leaf node or an interior node with exactly two children. Thus there is no need to create new leaf nodes in Pass-1. Second, the priorities of rules in Modified-BST may differ from the order specified by the node depths (the number of edges from the node to the root). So, an action needs the leaf-pushing only if it has a higher priority than its descendants. We call such a Pass-1 as *Modified Pass-1* and its pseudocode is shown in Fig. 6.

Step-3, bit merging. To further compress the partition, FFTA then performs *bit merging* on the output of Modified-ORTC. Bit merging is first proposed by bit weaving [6] as one of its building blocks. For each partition, bit merging groups rules into *chunks* according to their actions. Within each chuck, it repeatedly finds two rules that differ only in one bit in their match fields, and replaces them with a single rule where the differing bit in the match field is *. Consider *011 and *111 shown in Fig. 4(c) as an example, bit merging will merge them into **11 as Fig. 4(d) shows. The detail of bit merging is well discussed in [6]; we omit it in the paper.

Finally, by sorting aggregated rules in nondecreasing order of their amounts of *s, FFTA gets the aggregated rules and their priorities for this *prefix-permutable* partition.

z	m	a
1	0111	Fwd 1
2	*101	Fwd 2
3	1*0*	Fwd 3
4	1111	Fwd 1
5	*011	Fwd 1
6	1*1*	Fwd 3
7	****	Drop

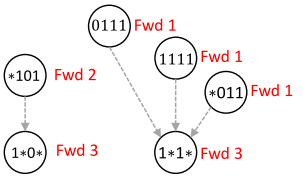
(a) the original table

z	m	a
1	0111	Fwd 1
2	*101	Fwd 2
4	1111	Fwd 1
5	*011	Fwd 1
3	1*0*	Fwd 3
6	1*1*	Fwd 3
7	****	Drop

(b) the reordered table

z	m	a
1	*101	Fwd 2
2	**11	Fwd 1
3	1***	Fwd 3
4	****	Drop

(c) the aggregated table



(d) the dependency graph with "****" omitted

Fig. 7. An example of increasing the chances of successfully aggregations with order rearrangements and aggregation-aware grouping.

4.2. *Aggregation-aware* Prefix-permutable *partition making*

Given a non-prefix flow table, there are many strategies to split it into *prefix-permutable* partitions. FFTA employs an aggregation-aware way: by rearranging some order-independent rules (i.e., rules with disjointed match fields) and grouping "similar" rules into the same partition, it significantly increases the chance of successfully aggregations.

Key idea. The intuition scheme for making *prefix-permutable* partitions is greedily cutting the original table into a list of consecutive partitions such that the concatenation of the partitions is the original table (Bit weaving [6] uses this scheme). For example, following such a scheme, the toy table shown in Fig. 7(a) will be split into 4 consecutive partitions of [0111,*101], [1*0*,1111], [*011], and [1*1*,****], where no partition can be aggregated further. Obviously, in this case, there is a chance to aggregate [0111,1111, *011], and [*1010, 1*1*]. However, they are grouped into distinct partitions. Note that 1*0*, 1111, and *011 are disjointed; it is safe to swap their orders/priorities. So, we can reorder them to split the original table into another two *prefix-permutable* partitions as Fig. 7(b) shows, which can be aggregated into 4 rules as Fig. 7(c) shows.

Inspired by the example, FFTA reorders some disjointed (i.e., order-independent) rules and groups those with the "similar" match fields into the same partition to increase the chances of successfully aggregations.

Similarity of rules. Given two match fields (i.e., two ternary strings), we use $\mathcal{H}$-distance to denote their degree of similarity, which is defined as the number of their disjointed symbols as Definition 2 shows. Such a definition is quite similar to the definition of *hamming distance*. However, in $\mathcal{H}$-distance, symbols might be *wildcards* and the distance between a wildcard and any other symbol is defined as *zero*. This distinguishes $\mathcal{H}$-distance from hamming distance. Obviously, if two rules are aggregatable, the $\mathcal{H}$-distance of their match fields must not exceed 1.

Definition 2 ($\mathcal{H}$-distance). For two equal length ternary strings, x and y, the $\mathcal{H}$-distance between them is the number of positions at which the corresponding symbols are disjointed, denoted as $\mathcal{H}(x, y)$. E.g., $\mathcal{H}(*, 0) \equiv \mathcal{H}(*, 1) \equiv \mathcal{H}(*, *) \equiv \mathcal{H}(1, 1) \equiv \mathcal{H}(0, 0) \equiv 0$, $\mathcal{H}(1, 0) \equiv 1$.

Observation 2. Given two equal length ternary strings, x and y, if they can be aggregated into one rule, then $\mathcal{H}(x, y) \leq 1$.

Observation 2 gives a necessary condition to check whether two rules are aggregatable. It also implies that, if adding a rule to a partition increases successfully aggregations, the added rule must can be aggregated with one of the pre-existent rules.

Theorem 2. *Given a set of ternary strings l and a ternary string x, where $l \cup [x]$ are prefix-permutable and l can be minimized to j rules, if $l \cup [x]$ can be aggregated to less than $j + 1$ rules, then* $\min_{y \in l} \mathcal{H}(x, y) \leq 1$.

Proof. Suppose $\min_{y \in l} \mathcal{H}(x, y) > 1$, say, x is disjointed with all ternary strings in l and there does not exist any ternary string that x can be aggregated with. Then x must not increase successfully aggregations of l. □

We further define $\mathcal{H}^L(x, l) \equiv \min_{y \in l} \mathcal{H}(x, y)$ and denote it as the distance between rule x and the rule set l. Accordingly, Theorem 2 implies Guideline 1 for making partitions.

Guideline 1. For each rule, group it into the nearest partition will increase the chance of successfully aggregation.

Making aggregation-aware partitions. As the disjointness of non-prefix rules is ubiquitous [15], there are multiple choices of making partitions with reordering. To guarantee the semantics of the flow table unchanged, the reordering should not change the relative orders of r_i and r_j if their match fields intersect. Thereafter, we define $m_i \wedge m_j$ to be the intersecting region (i.e., the common region) of m_i and m_j, e.g., $1000 \wedge 1001 = \emptyset$ and $10 * 1 \wedge 101* = 1011$. Accordingly, if $m_i \wedge m_j \neq \emptyset$ and $z_i < z_j$, the indexes of their partitions, denoted as $\pi(r_i)$ and $\pi(r_j)$, should satisfy: $\pi(r_i) \leq \pi(r_j)$. In consideration of the order-dependency relationships between rules can be built as a directed acyclic graph (DAG), FFTA makes *prefix-permutable* partitions by repeatedly checking rules in a topological ordering of their DAG and grouping rules into partitions following Guideline 1.

The pseudocode of how FFTA makes partitions is shown in Fig. 8. It is analogous to the algorithm of topological sorting described by Kahn [25]. At each turn, FFTA picks an "independent" rule (a node with no incoming edges) r_i from the DAG (Line 7). If there is no feasible *prefix-permutable* partitions for r_i, or the minimal distance between r_i and partitions is larger than a pre-defined threshold (Line 32–34), FFTA creates a new *prefix-permutable* partition for r_i (Line 9). Otherwise, FFTA assigns r_i to the nearest partition (Line 13). After r_i's partition is established, FFTA immediately updates the smallest feasible partition indexes for all r_i's successor in the DAG (Line 17). This guarantees that the requirements of $\pi(r_i) \leq \pi(r_j)$ will be satisfied to all r_j whose match field intersects with r_i and $z_i < z_j$.

As an example, consider the toy table shown in Fig. 7(a). Its DAG is as Fig. 7(d) shows (the "catch-all" rule **** is

```
1: function MKPARTITION(T)                          ▷ T is a list of n rules
2:     G ← Build the dependency graph for T          ▷ a DAG
3:     S ← Set of all rules with no incoming edges in G
4:     B ← [0, 0, ···]                               ▷ for each rule
5:     L ←an empty list                              ▷ the list of partitions
6:     while S is non-empty do
7:         remove r_i from S, where z_i = min{z_i | r_i ∈ S}
8:         pIndex ← FINDPARTITION(r_i, L, B[i])
9:         if pIndex = −1 then                       ▷ no partition is found
10:            Append new Partition [r_i] to tail of L
11:            pIndex ← L.length
12:        else
13:            Append r_i to Partition L[pIndex]
14:        end if
15:        for each rule r_j with an edge e from r_i to r_j do
16:            remove edge e from G
17:            B[j] ← max(B[j], pIndex)
18:            if r_j has no other incoming edges then
19:                Insert r_j into S
20:            end if
21:        end for
22:    end while
23:    return L
24: end function
25: function FINDPARTITION(r_i, L, j)
26:    pIndex ← −1; d ← Inf
27:    for k ← j to L.length do
28:        if P_k ∪ {r} is prefix-pe. and d ≤ H^L(r_i, P_k) then
29:            d ← H^L(r_i, P_k); pIndex ← k
30:        end if
31:    end for
32:    if d > d_threshold then
33:        pIndex ← −1
34:    end if
35:    return pIndex
36: end function
```

Fig. 8. The pseudocode of making *prefix-permutable* partitions in an aggregation-aware fashion.

omitted in the DAG). With MKPARTITION, FFTA will split it into two *prefix-permutable* partitions as Fig. 7(b) shows.

5. Incremental update incorporation with online-iFFTA

In most cases (more than 97% in our analysis), iFFTA can easily incorporates an update (i.e., insertion, deletion, or modification of a rule) to the aggregated table by (1) locating the involved partition, and (2) recomputing the affected regions. Unfortunately, as some rules might have been reordered when split into *prefix-permutable* partitions, a few insertions will involve the re-aggregation of multiple partitions if the inserted rules happen to introduce newly dependencies to the reordered rules. For this problem, iFFTA first computes the position that causes the minimal violations of dependency for the insertion, then installs a *virtual* rule for each violation to avoid the heavy re-aggregation. In the following, we will describe how iFFTA handles insertions, deletions, and modifications in detail.

5.1. Insertion

When inserting a new rule r_i to the original table T^o, we denote π^o to be the old partition scheme before the update, and π to be the new partition scheme after the update, respectively. For original rules in T^o, we further define $\bar{I}(r_i)$ to be the set of rules that cover r_i, and $\underline{I}(r_i)$ to be the set of rules that r_i covers, i.e., $\bar{I}(r_i) \equiv \{r_j \in T^o | z_j < z_i, m_i \wedge m_j \neq \emptyset\}$, $\underline{I}(r_i) \equiv \{r_j \in T^o | z_i < z_j, m_i \wedge m_j \neq \emptyset\}$. Obviously, to not change the table's semantics, the new partition index of r_i should satisfy

In Eqs. (1) and (2) as Section 4.2 discussed.

$$\max_{r_j \in \bar{I}(r_i)} \pi(r_j) \leq \pi(r_i), \text{ If } \bar{I}(r_i) \neq \emptyset \quad (1)$$

$$\min_{r_j \in \underline{I}(r_i)} \pi(r_j) \geq \pi(r_i), \text{ If } \underline{I}(r_i) \neq \emptyset \quad (2)$$

In the case that $\bar{I}(r_i)$ and $\underline{I}(r_i)$ are non-empty, we further define $\overline{\pi^o}(r_i)$ to be shorthand for $\max_{r_j \in \bar{I}(r_i)} \pi^o(r_j)$ and $\underline{\pi^o}(r_i)$ to be shorthand for $\min_{r_j \in \underline{I}(r_i)} \pi^o(r_j)$. Accordingly, there are 6 cases for r_i's insertion:

C1. $\bar{I}(r_i) = \emptyset$ **and** $\underline{I}(r_i) = \emptyset$;
C2. $\bar{I}(r_i) \neq \emptyset$ **and** $\underline{I}(r_i) = \emptyset$;
C3. $\bar{I}(r_i) = \emptyset$ **and** $\underline{I}(r_i) \neq \emptyset$;
C4. $\bar{I}(r_i) \neq \emptyset$ **and** $\underline{I}(r_i) \neq \emptyset$ **and** $\overline{\pi^o}(r_i) < \underline{\pi^o}(r_i)$;
C5. $\bar{I}(r_i) \neq \emptyset$ **and** $\underline{I}(r_i) \neq \emptyset$ **and** $\overline{\pi^o}(r_i) = \underline{\pi^o}(r_i)$;
C6. $\bar{I}(r_i) \neq \emptyset$ **and** $\underline{I}(r_i) \neq \emptyset$ **and** $\overline{\pi^o}(r_i) > \underline{\pi^o}(r_i)$.

In **C1**, r_i is disjointed with all existing rules; it is safe to directly insert r_i to any positions in the aggregated table. In **C2**, rules that intersect with r_i all have higher priorities than r_i; it is safe to directly insert r_i to any positions after $\overline{\pi^o}(r_i)$ in the aggregated table. In **C3**, rules that intersect with r_i all have lower priorities than r_i; it is safe to directly insert r_i to any positions before $\underline{\pi^o}(r_i)$ in the aggregated table. In **C4**, the partition indexes of rules covering r_i are smaller than that of rules covered by r_i; it is safe to directly insert r_i to any positions between partition $\overline{\pi^o}(r_i)$ and partition $\underline{\pi^o}(r_i)$.

Obviously, in **C1**–**C4**, iFFTA can simply make a new *prefix-permutable* partition for r_i, and directly inserts this partition into one of the possible positions. However, the cases of **C5** and **C6** are more complicated. To incorporate r_i without changing table semantics, some aggregated rules might be re-aggregated.

In **C5**, $\overline{\pi^o}(r_i) = \underline{\pi^o}(r_i)$. Let $P^o(r_i)$ be the set of old original rules in this partition, i.e., $P^o(r_i) \equiv \{r_j \in T^o | \pi^o(r_j) = \overline{\pi^o}(r_i)\}$. Obviously, iFFTA can incorporate r_i by making a re-aggregation of $\{r_i\} \cup P^o(r_i)$. The re-aggregation involves two parts: updating the aggregated Modified-BST and rerunning bit merging for the modified rules (i.e., only the modified chunks). If there exists a host node for r_i in $P^o(r_i)$'s Modified-BST, it means that $\{r_i\} \cup P^o(r_i)$ is still *prefix-permutable*. In such a case, the update of Modified-BST is analogous to the update of aggregated prefix rules; techniques designed for the updates of prefix rules (e.g., SMALTA [11] and FIFA [12]) can be used. In this paper, iFFTA adopts another simpler method—rerun Modified-ORTC for the subtree rooted from r_i's node. Otherwise, it indicates that $\{r_i\} \cup P^o(r_i)$ is not *prefix-permutable*, or the Modified-BST must be reconstructed. Accordingly, iFFTA treats $\{r_i\} \cup P^o(r_i)$ as a virtual flow table then makes a snapshot aggregation with FFTA.

In **C6**, the rule to be inserted (i.e., r_i) introduces new order-dependencies to some reordered rules, and these newly introduced dependency requirements are violated in π^o. Roughly, there are two possible solutions.

The first and the naive solution is to treat $\{r_i\} \cup \{r_j \in T^o | \underline{\pi^o}(r_i) \leq \pi^o(r_j) \leq \overline{\pi^o}(r_i)\}$ as a virtual table, then make a snapshot aggregation of it with FFTA. However, using such a solution, a lot of partitions will be repartitioned and reaggregated even if their rules are disjointed with r_i.

The second solution is to add *virtual* rules to remedy dependency violations introduced by the insertion. This design is motivated by Observation 3, which indicates: the impact of a violation of order-dependency can be offset by constructing and inserting a *virtual* rule before the violated rule.

Observation 3. Given two rules r_i: $\langle m_i, a_i, z_i \rangle$ and r_j: $\langle m_j, a_j, z_j \rangle$, where $z_i < z_j$ and $m_i \wedge m_j \neq \emptyset$, $\{r_i, r_j\}$ is equivalent to $\{\langle m_k, a_i, z_k \rangle, \langle m_j, a_j, z_i \rangle, \langle m_i, a_i, z_j \rangle\}$ where $m_k = m_i \wedge m_j$ and $z_k = z_i - 1$.

Suppose the rule to be inserted (i.e., r_i) were incorporated into a partition whose old index is p. Then, for each rule in set $\{r_j \in \bar{I}(r_i) | \pi^o(r_j) > p\}$, we need to construct and insert a *virtual* rule before or in the front of partition p. And for each rule in set $\{r_j \in \underline{I}(r_i) | \pi^o(r_j) < p\}$, we need to construct and insert a *virtual* rule before or in the front of that rule's partition. Therefore, the total number of *virtual* rules that need be added is $|\{r_j \in \bar{I}(r_i) | \pi^o(r_j) > p\}| + |\{r_j \in \underline{I}(r_i) | \pi^o(r_j) < p\}|$, which is a function of p. We denote the function as $f(p)$. Accordingly, by computing $\arg\min_p f(p)$, we can easily find the "optimal" partition for a given insertion. In addition, it should be noted that the insertion of r_i must fall into the case of **C5**, as well the insertion of *virtual* rules must fall into the case of **C3** or **C4**. So, in essence, this solution is to degrade a type-**C6** insertion to a type-**C5** with the cost of adding $\min_p f(p)$ *virtual* rules. It is applicable to the case in which $\min_p f(p)$ is small.

As both solutions have their own strengths and weaknesses, iFFTA makes a combination use of them. Once a type-**C6** insertion comes, if its $\min f(p)$ is smaller than a threshold (e.g., 10), iFFT employs the second solution; otherwise, iFFTA employs the first naive solution.

5.2. Deletion

As the deletion of rules would not introduce new dependency requirements, its operation is quite simple by contrast with the insertion. When deleting r_i from its aggregated partition, iFFTA first locates r_i's host node in the Modified-BST, then updates the involved subtree and reruns bit merging for the modified rules. In a few causes, a rule deletion will make an original MCS incomplete. Then iFFTA will update the entire MCS and break it into multiple nano MCSs.

In addition, if the to-be-deleted rule has caused *virtual* rules, all its *virtual* rules would be remove simultaneously.

5.3. Modification

For the modification of a rule, iFFTA treats it as an insertion followed by a deletion. This is also the way how commercial switches implement rule modifications [26].

6. Complexity analysis

The computation of FFTA mainly consists two computationally stages: (1) splitting the original table into *prefix-permutable* partitions, and (2) aggregating each *prefix-permutable* partition with Modified-ORTC and bit merging.

Let w be the number of bits within the match field. Both the calculation of $m_i \wedge m_j$ and the test of $W(x) \subseteq W(y) \vee W(y) \subseteq W(x)$ can be performed within $O(w)$. Consequently, both

the time complexity of constructing a dependency graph for n rules and making a topological sorting for an n-rule DAG would not exceed $O(wn^2)$. So, the worst-cast time complexity of splitting an n-rule table into *prefix-permutable* partitions with MKPARTITION is $O(wn^2)$. For each *prefix-permutable* partition, FFTA first constructs it as a Modified-BST, then uses the Modified-ORTC to aggregate. Obviously, for a partition with m rules, the number of nodes in the Modified-BST would not exceed wm. Again, since the complexity of the ORTC algorithm is linear in the number of nodes in the tree [10], the worst-cast time complexity of our ORTC-based aggregation is also $O(wm)$. After Modified-ORTC, FFTA further employs bit merging to merge rules that sharing the same action. The complexity of bit merging is proven to be $wk^{2\ln 3}$, where k is the number of rules in the input [6]. So, the worst-case time complexity of FFTA on the aggregation of each partition is $O(wm) + O(\max_{k\le m} wk^{2\ln 3}) = O(wm^{2\ln 3})$. Accordingly, the worst-case time complexity of FFTA on aggregating a table with n rules is $O(wn^2) + O(\max_{m\le n} m^{2\ln 3}) = O(wn^{2\ln 3})$.

As iFFTA degenerates into FFTA in the worst case, its worst-case time complexity is $O(wn^{2\ln 3})$ as well.

7. Discussion

7.1. About implementation

As both FFTA and iFFTA are software-based schemes, it is easy to implement flow table aggregation as an optimal service (FTAaaS, Flow Table Aggregation as a Service) on the controller[2]. In practice, table aggregation is not needed by all switches and all the time. Actually, only when a switch's flow table is running out of space, it prefers the aggregation of rules. So, the controller can enable FTAaaS for the switches that have limited TCAMs spaces; FFTA as well as iFFTA is called only when table aggregation is needed.

7.2. Limitation of usage

In the initially OpenFlow design [28], switches might follow the reactive flow programming model, in which a controller responding to traffic installs microflows matching every supported OpenFlow field for active flows. With this model, the number of flow coexisting on a switch is determined by the flow arrival and duration patterns. Accordingly, rules in a flow table are volatile and the effect of table aggregation is limited. In such a case, to make efficient use of the table, the controller can dynamically adapt the timeout setting of each flow entry with respect to flow patterns. Approaches like AHTM [29] follow this direction and might help. Indeed, recent study has proved that reactive programming of microflows is impractical for use outside of small deployments; while proactive population of flow tables which uses wildcard rules to cover the header space of all possible packets is preferred [27]. In this case, FFTA and iFFTA can help switches reduce the number of rules.

[2] If switches have local controllers [27], the aggregation can also be implemented at each switch.

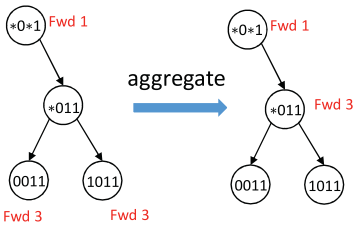


Fig. 9. The aggregation of rule 0011 and 1011 mixes their counters and conceals the death of either sub-flow that falls into 0011 or 1011.

7.3. Impact on OpenFlow protocol

Both FFTA and iFFTA deal with standard forwarding entries and require no modification of the Flow Table structure nor OpenFlow Protocol. However, as table aggregation is to merge rules together, an implicit side-effect is that the counters of multiple original rules are mixed up in the aggregated table. In the following, we sketch two typical impacts caused by aggregation and propose possible remedies.

Counters. In OpenFlow, each rule in the table (i.e., flow entry) has counters, which store the amounts of received packets and received bytes [1]. The statistical results might trigger the change of forwarding rules, like adding rules for firewall checking, or modifying some next-hops for load balancing. If table aggregation is employed, the controller only knows the aggregated count values, which might mislead the adjusting control.

Idle timeout. In some cases, a rule in the flow table might be assigned with an *idle_timeout*, indicating the maximum amount of idle time before it is expired by the switch [1]. In cases like flow based routing, if two rules are merged, the aggregated rule is unaware of whether a sub-flow is expired; then the death of that flow would be concealed, as the toy example in Fig. 9 shows.

Essentially, the side-effect of aggregation is caused because the mix of rule statistics has negative impacts on the evolution of network configuration. Note that, in practice, the forwarding rules in a table is generally generated by high level applications like routing, firewall, load balancer, monitoring, and so forth. According to the role of traffic statistics, each application might implicitly indicate whether its rules are aggregatable or not. For example, a rule that has an idle timeout setting or whose counter would immediately trigger firewall checking (or other stateful forwarding), should not be aggregated. By splitting each unaggregatable rule into a separate partition, FFTA and iFFTA can handle this easily. In other cases like load balance routing, rule aggregation is acceptable but the statistics of original rules need to be estimated. In SDN, the global knowledge of the network and fine-grained programmability of switches provide more flexible ways to make traffic monitoring and composition estimation. Lots of measurement systems and approaches are suggested by recent literatures [2,30–33]. With the help of these measurement systems and the global knowledge of aggregated statistics, the controller is able to infer the detail statistics of original rules from the aggregated statistics. However, the design of such an orchestrated system is still an open problem; we leave it as our future work.

Table 1
The information of forwarding tables in Stanford University Backbone Network.

Table	bbra,bbrb	boza,bozb	coza,cozb	goza,gozb	poza,pozb	roza,rozb	soza,sozb	yoza,yozb
Original size	1825,1620	1614,1453	184909,183376	1767,1669	1489,1434	1567,1483	184682,180944	4746,2592
AN(Action Num.)	61,40	25,26	42,41	20,20	18,17	17,15	48,39	77,48
MCS Num.	18,18	11,11	60022,60015	11,11	12,12	11,11	60015,60013	13,12
Aggregated size	691,662	180,156	47973,47947	147,130	103,88	97,85	47991,47956	184,115
Agg. ratio	37.9%,40.9%	11.2%,10.7%	25.9%,26.2%	8.3%,7.8%	6.9%,6.1%	6.2%,5.7%	26.0%,26.5%	3.9%,4.4%

8. Performance evaluation

In this section, we use prefix rules from Stanford University Backbone Network [16] and synthetic non-prefix rules from ClassBench [17] (with real parameters) to evaluate the effectiveness and efficiency of FFTA and iFFTA. Extensive experimental results demonstrate that:

1. On the aggregation of *prefix-permutable* partitions, FFTA is about 200 × faster than bit weaving, while using much less memories and sharing the same compression ratio.
2. On incorporating an update to its *prefix-permutable* partition, iFFTA is about 3 × faster than FFTA with an acceptable loss of compression ratio.
3. On the aggregation of entire flow tables, FFTA is significant better than bit weaving on the average compression ratio with the improvement up to 48%.
4. On incorporating updates into tables, by using the order-independences of rules, iFFTA greatly simplifies the revision and computation of the aggregated table—more than half of insertions can be directly incorporated without any recomputation/modification of the aggregated rules, and all other updates can be incorporated with the recomputation of only one *prefix-permutable* partition per update.

8.1. Methodology

Implementation. As bit weaving is the best reported non-prefix aggregation scheme at current, we mainly use it as the baseline in the paper. To simplify the comparison, neither bit weaving nor our solution uses other optimal schemes like *Redundancy Removal* [6]. In tests, all algorithms are implemented in Python, where the implementation of bit weaving directly uses the code released by its authors [34] as the core (also written in Python). All experiments are carried out by Python 3.2.3 on a PC running 64-bit Ubuntu 12.04 server with 6G memory and a single Intel i7-930 CPU. All algorithms only use a single processor core.

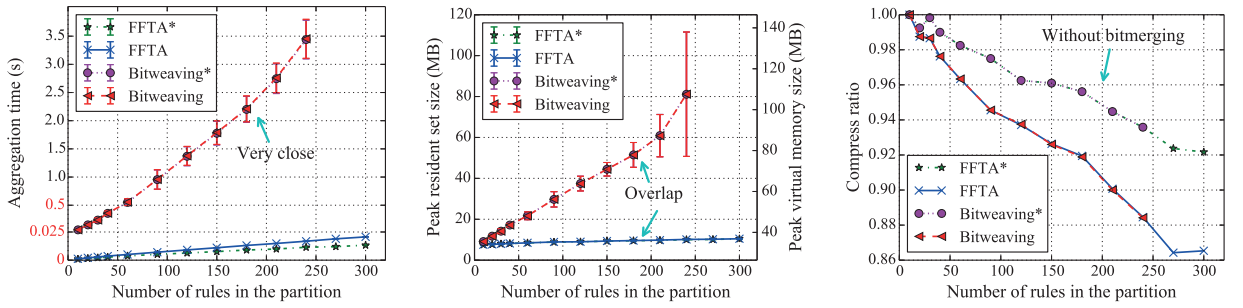
Data sets. We use two types of rules in the tests: one is from Stanford University Backbone Network [16] and the other is from ClassBench [17] (with real parameters).

Firstly, as both bit weaving and our solution build on the aggregation of a single *prefix-permutable* partition, where rules in the partition can be simultaneously permuted into prefix format, we directly use publicly available prefix rules to synthesize *prefix-permutable* partitions to test their performances on the aggregation of partitions. These prefix rules are collected from Stanford University Backbone Network [16], in which 14 operational zone Cisco routers connect 2 backbone routes (named *bbra* and *bbrb*) via 10 Ethernet switches, and the 2 backbone routes connect Stanford to the outside world in turn. Recall that our *prefix-permutable* partitions are usually incomplete, we ignore the default route (i.e., the all-* entry) in each input prefix table. Table 1 details the information of the 16 routing tables, where *Original Size*, *AN*, and *MCS Num.* denote the amounts of original rules, actions, and maximal complete subtrees (MCS) for each table, respectively.

We synthesize *prefix-permutable* partitions by randomly selecting prefix rules from one of the tables shown in Table 1. Similarly, we synthesize incremental updates by randomly choosing a group of rules from a given partition to insert, or to delete. Previous statistics have shown that the size of *prefix-permutable* partitions ranges from several to hundreds in today's non-prefix classifiers [6]. In common with this, we let the size of synthetic partitions ranges from 10 to 300 with step 10 in our experiments. For each partition size, we randomly generate 20 partitions. We use all the 16 routing tables as the input table to drive the tests. Their results imply the consistent observations and the figures shown in the Section 8.2 are the cases of using *bbrb*'s table.

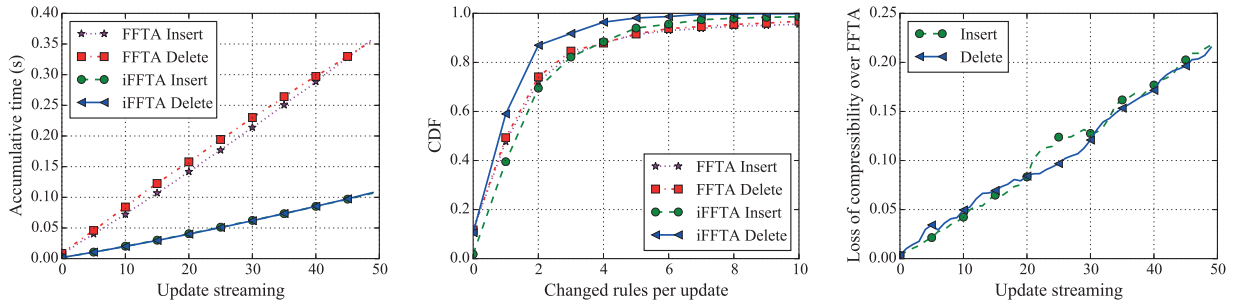
Secondly, to evaluate FFTA and iFFTA on the aggregation of entire flow tables, we further run simulations on synthetic tables/classifiers from ClassBench [17]. Classbench provides 12 parameter files (acl 1-5, fw 1-5, ipc 1-2) abstracted from 12 real filter sets, which fall into 3 types of formats: access control list (ACL), firewall (FW), and IP chain (IPC). To study the impacts of classifier size, we suppose that a table/classifier would involve 100, 500, or 1K original rules. For each parameter file and each table size, we generate 20 tables. The action type of each rule in a table is randomly selected from $1, 2, \ldots, AN$. In our tests, *AN* is set with 1, 2, and 4. To stress test the sensitivity of our algorithms to the number of actions, we also test the case where each rule has a unique action (i.e., $AN = \infty$). For each *non-wildcard* rule, we transfer it into *wildcard* rule(s) by encoding its port number ranges into the minimum prefixes [35]. We just use the four fields of source IP, destination IP, source port, and destination port as the rule's match fields here. Finally, we get the synthetic flow table by (1) removing all duplicated rules, and (2) sorting remaining rules according to the *weight*s of their matching fields. In this paper, the *weight* of a length-K ternary string x is defined as $\sum_{j=0}^{K-1} eval(x[j]) \times 3^j$, where $eval(0) = 0$, $eval(1) = 1$, and $eval(*) = 2$.

Metrics. Let T be an original non-prefix table and $A(T)$ be its aggregated table with aggregation algorithm A employed. Then the *Compression ratio* of A on T is defined as $\frac{|A(T)|}{|T|}$, where $|A(T)|$ and $|T|$ denote their numbers of rules, respectively. Further, for another aggregation algorithm $A^\dagger$, the improvement ratio of $A^\dagger$ over A on T is defined by $\frac{|A(T)|-|A^\dagger(T)|}{|A(T)|}$.



(a) Time cost over partition size (b) Memory cost over partition size (c) Compression ratio over partition size

Fig. 10. FFTA outperforms bit weaving at the running time and memory without loss of aggregation effectiveness. Besides, our accelerated version of bit merging does not increase the demands of memory.



(a) Accumulative time cost (b) The CDF of changed rules per update (c) Loss of compressibility over FFTA

Fig. 11. iFFTA is about 3 × faster than FFTA on incorporating updates.

8.2. *On* Prefix-permutable *partitions*

As Section 3 has shown, the core of FFTA is the aggregation of each *prefix-permutable* partition. So, we firstly evaluate the effectiveness and efficiency of FFTA and iFFTA on aggregating *prefix-permutable* partitions comprehensively.

Snapshot aggregation. Fig. 10 shows the performance of FFTA on the snapshot aggregation of *prefix-permutable* partitions. The label with a * denotes the case where bit merging is disabled, e.g., FFTA* and Bitweaving*. The gap between the curves of FFTA and FFTA* shown in Fig. 10 indicates: though the worst-case time complexity of bit merging is $O(wn^{2\ln 3})$, its average running time grows linearly with the partition size in practice. Accordingly, the average running time of FFTA is quite natural and logical to grow linearly with the partition size, because its prefix-based aggregation technique, *Modified-ORTC*, is $O(wm)$ (refer to Section 6). So does bit weaving—its prefix-based aggregation technique, *weighted one-dimensional prefix list minimization algorithm* (a dynamic programming algorithm), is $O(wm)$ as well [5]. However, FFTA has a much smaller constant factor. Extensive results imply that FFTA is about 200 × faster than bit weaving on average. For instance, FFTA costs less than 21 ms to aggregate a partition with 300 rules while bit weaving needs about 3.5 s to aggregate a partition with 240 rules. Recall that, both bit weaving and FFTA split each flow table into *prefix-permutable* partitions to perform aggregation. When excluding the variance of their time costs on making partitions, FFTA would be about 200 × faster than bit weaving on the aggregation of entire tables as well.

From Fig. 10(b), we observe that FFTA costs much less memories than bit weaving, where the two *y-axiss* denote the peak usage of physical memory and the virtual memory, respectively. In tests, once the partition size is larger than 250, the process of bit weaving is always put in to *Disk Sleep* state and becomes a zombie process. So we only test partitions with no more than 240 rules for bit weaving.

Fig. 10 (c) shows the average compression ratios of FFTA and bit weaving on partitions. It implies that FFTA achieves the same compression ratio with bit weaving on a given partition. This is because FFTA's Modified-ORTC shares the same compression ratio with the *weighted one-dimensional prefix list minimization algorithm* that bit weaving uses; they both aggregate the *prefix-permutable* rules into the minimized number of prefix rules (refer to [24] for details).

Incremental update. To evaluate the performance of FFTA and iFFTA on incorporating updates to an aggregated partition, we randomly select 50 rules from a synthetic partition, then add them to the remainder partition, or delete them from the original partition one by one. Fig. 11 shows the results of the case where the original partition is made up of the *bbrb*'s first 75 prefixes and last 75 prefixes.

Fig. 11 (a) shows the average computing time of FFTA and iFFTA on incorporating updates. It indicates that (1) iFFTA is about 3 × faster than FFTA, and (2) iFFTA as well as FFTA has the similar time complexity on insertion and deletion. For example, iFFTA costs less than 2.5 ms on incorporating an update while FFTA needs about 7.5 ms. The results also imply that, though iFFTA shares the same worst-case time complexity with FFTA (refer to Section 6), it is more efficient in

Table 2
The impact of table property on aggregation effectiveness.

	Orig. table size (avg.)			Compression ratio of FFTA (avg.)										
				$AN = 1$			$AN = 2$			$AN = 4$			$AN = \infty$		
	100	500	1K	100	500	1K	100	500	1K	100	500	1K	100	500	1K
acl1	136	655	1266	.885	.672	.642	.944	.810	.781	.969	.876	.856	.997	.944	.930
acl2	222	981	1882	.845	.642	.645	.880	.727	.736	.907	.778	.784	.922	.821	.835
acl3	182	894	1718	.973	.920	.896	.981	.944	.930	.988	.958	.950	.991	.973	.967
acl4	172	829	1651	.970	.927	.881	.984	.951	.917	.989	.967	.939	.996	.980	.961
acl5	124	592	1123	.949	.880	.804	.964	.923	.861	.974	.945	.893	.983	.968	.924
fw1	348	1450	3006	.840	.775	.757	.888	.823	.811	.921	.853	.837	.951	.883	.866
fw2	180	856	1788	.705	.751	.859	.798	.814	.908	.840	.850	.934	.891	.889	.962
fw3	239	1199	2354	.748	.783	.776	.840	.848	.837	.896	.883	.875	.942	.921	.906
fw4	588	2735	4638	.912	.816	.752	.947	.870	.815	.959	.899	.848	.975	.931	.889
fw5	239	1054	2074	.791	.760	.772	.879	.847	.851	.919	.889	.896	.978	.937	.943
ipc1	140	660	1318	.894	.831	.778	.933	.904	.868	.948	.938	.914	.972	.977	.962
ipc2	85	310	662	.491	.418	.411	.626	.548	.538	.687	.625	.609	.782	.703	.693

practice. This is because iFFTA only recomputes the affected regions for each update, while FFTA performs a fresh snapshot aggregation of the entire partition.

We also count the change of aggregated rules for the incorporation of each original update. Their distributions are shown in Fig. 11(b). The results imply that there is very little difference between iFFTA and FFTA. We also test the loss of compressibility for iFFTA (i.e., the improvement ratio of FFTA over iFFTA), which assesses how much additional compression would be achieved if we make a fresh snapshot aggregation for each update. The observations from various partitions indicate that the loss of compressibility is always small. However, the relationship of the curves of insertion and deletion is tightly related to the test instance. The one shown in Fig. 11(c) is the result of the test case we mentioned before.

Further, we rerun the experiments with all other 15 prefix tables and with different synthetic strategies; the results imply the consistent conclusions. In addition, the snapshot compression ratio of each prefix table with the default rule excluded is shown in Table 1 as well.

8.3. On flow tables

As the above subsection has shown, FFTA shares the same compression ratio with bit weaving on the snapshot aggregation of *prefix-permutable* partitions. In this subsection, we further investigate how the proposed aggregation-aware reordering scheme helps each partition's aggregation, and how it impacts the incorporation of updates.

Impact of table property. Table 2 shows the effectiveness of FFTA on the aggregation of synthesized tables from ClassBench [17], where each value denotes the average compression ratio of the 20 test cases (recall that, we synthesize 20 tables for each parameter file and each scale value). The 2nd–4th columns show the average amounts of rules in each synthetic table. Because of the range encoding, the table sizes are generally larger than the parameter value that we use to drive ClassBench's generator (i.e., 100, 500, 5K). For example, a "100-rule" table generated with *acl2*, will expand to about 200 *wildcard* rules. However, there is a notable exception of *ipc2*, the amount of whose encoded *wildcard* rule is less than the scale parameter. This is because many rules generated with *ipc2* only differ in the protocol and additional fields (i.e., the 5th and 6th fields). When only the first four fields are considered, *ipc2* contains a lot of duplicated rules, which are removed.

From the results, we find that a table's compression ratio is closely related with its type, size, and possible action numbers. In most cases, FFTA will get a better aggregation on the table with a larger size, or with less action numbers. We think, it is mainly because both the raise of flow table size and the reduction of possible action number might increase the chance of successfully aggregation (i.e., there are more redundancies). But *fw2* is a notable exception, on which we get a worse compression ratio with the table size growing. We argue that, on *fw2*'s tables, the increase of successfully aggregation is slower than the increment of table size. Another observation is that, FFTA always gets successfully aggregations in tests, even when each rule is assigned with a unique action (i.e., $AN = \infty$). We think there are two reasons. Firstly, if the original rule from ClassBench specifies a *non-wildcard* port range, it creates multiple wildcard rules after range encoding. As these children rules share the same action, there are chances for aggregation. Secondly, the match fields of rules may intersect. Once a rule is completely covered by other rules in the same partition, it will be removed during the aggregation.

Effectiveness of aggregation-aware reordering. Previous literatures [36,37] have shown that order-independent rules in a table might have non-deterministic orders in practice. To evaluate the effectiveness of the aggregation-aware partition-making scheme used in FFTA, we randomly shuffle the order-independent rules in each synthetic table, and compare the compression ratio of FFTA against bit weaving—which does not reorder any rule when making partitions. Table 3 shows the average improvement ratio of FFTA on tables with varied parameters. It indicates that FFTA always gets a better average compression ratio than bit weaving. We also observe that the improvement ratio generally grows with the flow table size increasing or with the action number decreasing. This implies that the proposed aggregation-aware scheme is more effective on tables with more redundancy. Take *ipc2*'s tables as an example, with the table scale parameter increases from 500 to 1K, the average improvement ratio of the table with 4 types of actions also increases from 17.6% to 26.9%. Again, with the action number decrease

Table 3
The effectiveness of FFTA's aggregation-aware partition-making scheme: the (avg.) improvement ratio over bit weaving.

	$AN = 1$			$AN = 2$			$AN = 4$			$AN = \infty$		
	100	500	1K	100	500	1K	100	500	1K	100	500	1K
acl1	.081	.251	.315	.042	.138	.188	.024	.088	.119	.002	.040	.057
acl2	.094	.268	.293	.086	.203	.213	.066	.165	.174	.062	.137	.132
acl3	.020	.068	.091	.015	.048	.062	.009	.038	.045	.008	.025	.031
acl4	.024	.063	.104	.013	.044	.075	.009	.029	.054	.004	.019	.037
acl5	.045	.112	.188	.033	.073	.133	.025	.052	.103	.017	.031	.074
fw1	.111	.197	.225	.079	.157	.175	.057	.132	.151	.037	.106	.126
fw2	.207	.190	.077	.149	.144	.055	.117	.119	.045	.086	.094	.033
fw3	.139	.184	.202	.086	.127	.146	.056	.097	.110	.036	.065	.082
fw4	.074	.171	.232	.046	.122	.173	.035	.094	.142	.023	.066	.104
fw5	.103	.202	.201	.059	.126	.129	.044	.088	.086	.011	.047	.043
ipc1	.078	.144	.194	.055	.081	.116	.047	.052	.075	.026	.023	.035
ipc2	.255	.395	.486	.155	.250	.344	.092	.176	.269	.053	.105	.183

Table 4
The impact of reordering on updates: statistics of insertion.

	100					500					1K				
	C1–C4	C5	C6	*avg*	*max*	C1–C4	C5	C6	*avg*	*max*	C1–C4	C5	C6	*avg*	*max*
acl1	.962	.017	.021	1.6	3	.910	.067	.023	1.8	7	.896	.086	.018	2.6	9
acl2	.981	.016	.002	2.0	2	.916	.078	.006	1.8	5	.894	.097	.009	1.4	3
acl3	.986	.006	.008	1.0	1	.972	.007	.021	2.6	24	.961	.020	.019	1.6	8
acl4	.988	.006	.006	3.5	5	.973	.009	.018	1.6	5	.951	.023	.027	2.1	20
acl5	.992	.008	.000	.0	0	.951	.035	.014	1.6	4	.910	.070	.020	1.3	3
fw1	.955	.038	.007	2.6	7	.939	.056	.005	4.4	27	.938	.057	.006	9.5	33
fw2	.899	.089	.012	1.0	1	.904	.089	.007	1.6	5	.972	.023	.005	1.1	2
fw3	.939	.054	.007	1.2	2	.922	.072	.006	8.5	22	.919	.076	.005	9.7	23
fw4	.982	.015	.003	4.8	16	.966	.029	.005	2.5	12	.949	.047	.004	5.2	78
fw5	.938	.058	.004	5.0	5	.930	.068	.002	3.4	5	.909	.086	.004	2.0	5
ipc1	.964	.025	.011	1.3	2	.925	.058	.017	2.0	8	.896	.082	.022	1.9	10
ipc2	.659	.341	.000	.0	0	.482	.516	.002	1.0	1	.557	.438	.005	1.1	2

to 2, the improvement ratio increases to 34.4%. However, *fw2* is a notable "exception", on which the average improvement ratio decreases with the table size growing. Recall that, *fw2*'s compression ratio decreases with its table size growing (see Table 2). That is to say, the compressibility of *fw2* decreases on lager tables. Consequently, it is quite natural and logical for the reduction of *fw2*'s improvement ratio.

Essentially, our aggregation-aware reordering scheme is still heuristic. It does not guarantee to outperform the non-reorder scheme used by bit weaving. For example, on the aggregation of *acl2*'s tables with 2 actions, the improvement ratio is less than 0 (≈ -0.04) in the worst-case as Fig. 12 shows. We investigate all the tests and find this happens only in rare cases (*ratio* $< 0.5\%$) and the improvement ratio is always lager than -0.1. That is to say, the proposed aggregation-aware scheme gets significant improvements in most cases. In practice, the controller can simply run both schemes for each snapshot aggregation and choose the aggregated table that wins.

Impact on updates. As discussed in Section 5, FFTA improves the aggregation effectiveness with the risky of complicating the insertion updates. To study the impacts, we randomly pick out 10% wildcard rules from each table, and insert them back to the remaining tables which have been aggregated by FFTA. Table 4 shows the statistics of the composition of these insertions. Obviously, most of the insertions fall into **C1–C4** (*ipc2*: $\approx$ 50%, all others: $\approx$ 90%). It means that most of inserted rules can be directly incorporated into the aggregated table without any re-aggregation. When performing such insertions, the mainly computation is to find a feasible position and the time complexity does not exceed $O(wn)$. Also, once a rule is directly inserted, its deletion will not cause any re-aggregation; the deletion operation can be performed in $O(1)$. Moreover, less than 3% of insertions belong to **C6**. For these type-**C6** insertions, we further investigate the amounts of *virtual* rules that will be added when using the second solution (i.e., add *virtual* rules) for their incorporations. The columns labeled *avg* and *max* in Table 4 show the average and maximum requirements of *virtual* rules, respectively. Results indicate that most of insertions in **C6** cause less than 10 *virtual* rules. That is to say, the price of FFTA's reorder-enabled partition-making scheme is quite small in practice.

9. Related work

Prefix aggregation. The issue of prefix aggregation have received considerable attentions from the research community over the last few years. Draves et al. [10] designed an offline algorithm called ORTC to generate the compressed IP routing table, which is proved to be optimal (means the number of entries in the generated table is minimized). Based on ORTC, online algorithms like SMALTA [11] and FIFA [12], are present to achieve fast incremental updates of aggregated tables with a sacrifice of compression effectiveness or

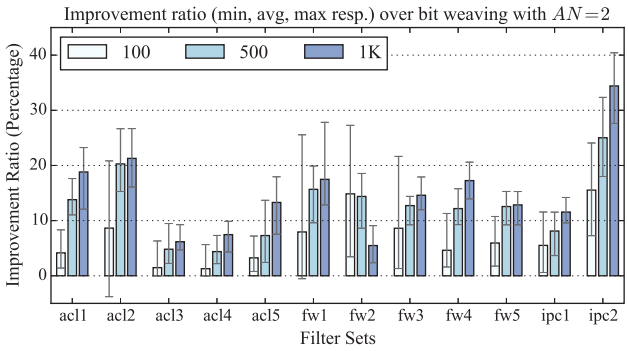


Fig. 12. A case study of the improvement ratio.

recomputing time. Although the aggregation of prefix rules is different from that of flow table, its techniques inspire our design. In addition, we employ a variant of ORTC on the aggregation of each *prefix-permutable* partition.

Non-prefix aggregation. The problem studied here is more similar to the aggregation of TCAM/non-prefix rules. McGeer and Yalagandula [4] formulated the TCAM rulesets minimization into a Boolean optimization problem. But their algorithms are either inefficient or unpractical for flow table aggregation. Liu et al. designed TCAM razor [5] and bit weaving [6] for non-prefix classifier aggregation. TCAM razor compresses multi-field classifiers by constructing a series of intermediate one-dimensional prefix classifiers. The method it employs only produces prefix classifiers and may miss some opportunities for compression. Bit weaving is excellent for offline aggregation, but impractical for dynamic networks, since a global, inefficient recomputing (re-aggregate a whole partition or even the whole flow table) is needed once a rule is updated. Our FFTA shares the similar basic idea and achieves the same compression ratio on the aggregation of each partition with bit weaving, but FFTA is more efficient (about 200 $\times$ faster with less memory usage) and friendlier to table updates. Moreover, FFTA significantly outperforms bit weaving on the aggregation of whole tables by making partition in an aggregation-aware manner.

Also, there are considerable literatures [38–42] focusing on designing range encoding schemes to transfer range based firewall or ACL rules into *wildcard* rules supported by TCAMs. In current OpenFlow [1], match fields are limited to be wildcards, so FFTA and iFFTA proposed for flow table aggregation do not face the problem of range encoding. Nonetheless, some high level applications like firewall might prefer to express their forwarding policies with range based rules. In these cases, the controller needs to pre-encode these ranges into wildcard rules, then calls FFTA and iFFTA if needed. It should be noted that, when performing the range encoding, these proposed approaches that require the modification of TCAM hardware [40,42], can not help.

Flow table evolution. In OpenFlow 1.0 [1], the first and the most widely supported version, each switch is abstracted as a single table of match-action rules. Such a simple model helps the OpenFlow protocol be popular quickly, as existing switches can support it with little change. However, this model is costly in use because a single table needs to store every combination of headers. Accordingly, the Multiple Match Table (MMT) model [1,2], as well as Reconfigurable Match Table (RMT) model [43] and P4 language [3], is introduced to make efficient use of switch resources. In essence, the building block of MMT as well as RMT is still a set of narrower single tables. So, if needed, the controller can also employ FFTA and iFFTA to aggregate them.

Others. More recently, Palette [8] and *One Big Switch* abstraction [9] have proposed the schemes of decomposing a flow table into subtables and distributing them among the paths to reduce the demands of flow table space for each switch. CacheFlow [7] uses rule caching techniques to virtualize the physical TCAMs to get the illusion of an infinite rule table. Similar to CacheFlow, AHTM (Adaptive Hard Timeout Method) [29] makes efficient utilization of the physical flow table by optimizing the timeouts of rules. While orthogonal to our work, all those works may be benefited since fewer rules would need to be distributed or cached.

10. Conclusion

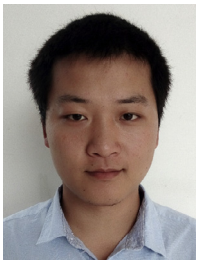
Flow table aggregation is a promising direction in reducing the requirements of TCAMs for SDN switches as it requires no modification to switch hardware or OpenFlow protocol. In this paper, we proposed FFTA and iFFTA to provide practical flow table aggregations. FFTA is a snapshot aggregation technique sharing the same high-level idea with the state-of-the-art scheme. However, because of the two major improvements it made, FFTA significantly outperforms the state-of-the-art scheme on both efficiency (about 200 $\times$ faster) and effectiveness (up to 48% improvements of the average aggregation ratio). Based on FFTA, iFFTA is designed to efficiently incorporate incremental updates to the aggregated tables. By using the order-independences of rules, iFFTA greatly simplifies the update—more than half of insertions can be directly incorporated without any recomputation/modification of the aggregated rules, and all other updates can be incorporated with the recomputation of only one *prefix-permutable* partition per update. On incorporating an update to its aggregated partition, by using the well-designed structural information of the aggregated rules, iFFTA further accelerates the recomputation/update about 3 $\times$ (compared with recalling FFTA) with an acceptable loss of effectiveness. Accordingly, FFTA and iFFA can cooperate on aggregating flow table; controllers should implement table aggregation as an optimal service.

Acknowledgments

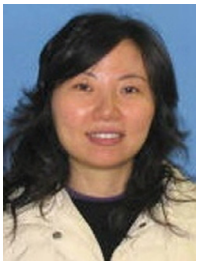
We thank the anonymous reviewers and editors for useful feedback. This work was supported in part by the 973 Program under Grant No. 2013CB329103, the 863 Program under Grant No. 2015AA015702 and the National Natural Science Foundation of China under Grant No. 61271171.

References

[1] O. N. Foundation, Openflow switch specification, (https://www.opennetworking.org, Last accessed: 05 Feb 2015).
[2] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, S. Uhlig, Software-defined networking: A comprehensive survey, Proc. IEEE 103 (1) (2015) 14–76, doi:10.1109/JPROC.2014.2371999.
[3] P. Bosshart, P4: Programming protocol-independent packet processors, SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95, doi:10.1145/2656877.2656890.
[4] R. McGeer, P. Yalagandula, Minimizing rulesets for tcam implementation, in: Proceedings of the IEEE INFOCOM, 2009, pp. 1314–1322, doi:10.1109/INFCOM.2009.5062046.
[5] A.X. Liu, C.R. Meiners, E. Torng, Tcam razor: a systematic approach towards minimizing packet classifiers in tcams, IEEE/ACM Trans. Netw. 18 (2) (2010) 490–500, doi:10.1109/TNET.2009.2030188.
[6] C.R. Meiners, A.X. Liu, E. Torng, Bit weaving: a non-prefix approach to compressing packet classifiers in tcams, IEEE/ACM Trans. Netw. 20 (2) (2012) 488–500.
[7] N. Katta, O. Alipourfard, J. Rexford, D. Walker, Infinite cacheflow in software-defined networks, in: Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN), 2014, pp. 175–180, doi:10.1145/2620728.2620734.
[8] Y. Kanizo, D. Hay, I. Keslassy, Palette: Distributing tables in software-defined networks, in: Proceedings of the IEEE INFOCOM, 2013, pp. 545–549, doi:10.1109/INFCOM.2013.6566832.
[9] N. Kang, Z. Liu, J. Rexford, D. Walker, Optimizing the "one big switch" abstraction in software-defined networks, in: Proceedings of the ACM CoNEXT, 2013, pp. 13–24.
[10] R. Draves, C. King, S. Venkatachary, B. Zill, Constructing optimal ip routing tables, in: Proceedings of the IEEE INFOCOM, vol. 1, 1999, pp. 88–97, doi:10.1109/INFCOM.1999.749256.
[11] Z.A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, P. Francis, Smalta: practical and near-optimal fib aggregation, in: Proceedings of the ACM CoNEXT, 2011, pp. 29:1–29:12.
[12] Y. Liu, B. Zhang, L. Wang, Fifa: Fast incremental fib aggregation, in: Proceedings of the IEEE INFOCOM, 2013, pp. 1–9, doi:10.1109/INFCOM.2013.6566913.
[13] D.A. Applegate, G. Calinescu, D.S. Johnson, H. Karloff, K. Ligett, J. Wang, Compressing rectilinear pictures and minimizing access control lists, in: Proceedings of the ACM-SIAM SODA, 2007, pp. 1066–1075.
[14] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, D. Walker, Abstractions for network update, in: Proceedings of the ACM SIGCOMM, 2012, pp. 323–334, doi:10.1145/2342356.2342427.
[15] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, P. Eugster, Saxpac (scalable and expressive packet classification), in: Proceedings of the ACM SIGCOMM, 2014, pp. 15–26, doi:10.1145/2619239.2626294.
[16] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, S. Whyte, Real time network policy checking using header space analysis, in: Proceedings of the USENIX NSDI, 2013, pp. 99–112.
[17] D.E. Taylor, J.S. Turner, Classbench: A packet classification benchmark, IEEE/ACM Trans. Netw. 15 (3) (2007) 499–511, doi:10.1109/TNET.2007.893156.
[18] A. Lara, A. Kolasani, B. Ramamurthy, Network innovation using openflow: A survey, Commun. Surv. Tutor. IEEE 16 (1) (2014) 493–512, doi:10.1109/SURV.2013.081313.00105.
[19] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74, doi:10.1145/1355734.1355746.
[20] N. Foster, R. Harrison, M.J. Freedman, C. Monsanto, J. Rexford, A. Story, D. Walker, Frenetic: A network programming language, SIGPLAN Not. 46 (9) (2011) 279–291, doi:10.1145/2034574.2034812.
[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker, Composing software-defined networks, in: Proc. USENIX NSDI, 2013, pp. 1–14.
[22] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, S. Shenker, Nox: Towards an operating system for networks, SIGCOMM Comput. Commun. Rev. 38 (3) (2008) 105–110, doi:10.1145/1384609.1384625.
[23] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar, Can the production network be the testbed? in: Proceedings of the USENIX OSDI, 2010, pp. 1–6.
[24] S. Luo, H. Yu, L.M. Li, Fast incremental flow table aggregation in sdn, in: Computer Communication and Networks (ICCCN), 2014 23rd International Conference on, 2014, pp. 1–8, doi:10.1109/ICCCN.2014.6911781.
[25] A.B. Kahn, Topological sorting of large networks, Commun. ACM 5 (11) (1962) 558–562, doi:10.1145/368996.369025.
[26] X. Jin, H.H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, R. Wattenhofer, Dynamic scheduling of network updates, in: Proceedings of the ACM SIGCOMM, 2014, pp. 539–550, doi:10.1145/2619239.2626307.
[27] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, M. Casado, The design and implementation of open vswitch, in: Proceedings of the USENIX NSDI, 2015, pp. 117–130.
[28] M. Casado, M.J. Freedman, J. Pettit, J. Luo, N. McKeown, S. Shenker, Ethane: Taking control of the enterprise, in: Proceedings of the ACM SIGCOMM, 2007, pp. 1–12, doi:10.1145/1282380.1282382.
[29] L. Zhang, R. Lin, S. Xu, S. Wang, AHTM: Achieving efficient flow table utilization in software defined networks, in: Proceedings of the IEEE Global Communications Conference (GLOBECOM), 2014, pp. 1897–1902.
[30] S. Shirali-Shahreza, Y. Ganjali, Flexam: Flexible sampling extension for monitoring and security applications in openflow, in: Proceedings of the 2nd Workshop on Hot Topics in Software Defined Networking (HotSDN), 2013, pp. 167–168, doi:10.1145/2491185.2491215.
[31] Y. Zhang, An adaptive flow counting method for anomaly detection in sdn, in: Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies(CoNEXT), 2013, pp. 25–30, doi:10.1145/2535372.2535411.
[32] M. Moshref, M. Yu, R. Govindan, A. Vahdat, Dream: Dynamic resource allocation for software-defined measurement, in: Proceedings of the ACM SIGCOMM, 2014.
[33] Y. Yu, C. Qian, X. Li, Distributed and collaborative traffic monitoring in software defined networks, in: Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN), 2014, pp. 85–90, doi:10.1145/2620728.2620739.
[34] C.R. Meiners, The implementation of 1-dimensional weighted prefix minimization (python), (http://www.cse.msu.edu/~meinersc/suri.py, Last accessed: 05 Feb 2015).
[35] S. Suri, T. Sandholm, P. Warkhede, Compressing two-dimensional routing tables, Algorithmica 35 (4) (2003) 287–300, doi:10.1007/s00453-002-1000-7.
[36] H. Song, J.S. Turner, Fast Filter Updates for Packet Classification using TCAM, in: Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM), 2006, pp. 1–5, doi:10.1109/GLOCOM.2006.342.
[37] X. Wen, C. Diao, X. Zhao, Y. Chen, L.E. Li, B. Yang, K. Bu, Compiling minimum incremental update for modular sdn languages, in: Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking(HotSDN), 2014, pp. 193–198, doi:10.1145/2620728.2620733.
[38] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, A. Shukla, Packet classifiers in ternary cams can be smaller, SIGMETRICS Perform. Eval. Rev. 34 (1) (2006) 311–322, doi:10.1145/1140103.1140313.
[39] A. Bremler-Barr, D. Hendler, Space-efficient tcam-based classification using gray coding, in: Proceedings of the IEEE INFOCOM, 2007, pp. 1388–1396, doi:10.1109/INFCOM.2007.164.
[40] C.R. Meiners, A.X. Liu, E. Torng, Topological transformation approaches to optimizing tcam-based packet classification systems, in: Proceedings of the ACM SIGMETRICS, 2009, pp. 73–84, doi:10.1145/1555349.1555359.
[41] R. Wei, Y. Xu, H. Chao, Block permutations in boolean space to minimize tcam for packet classification, in: Proceedings of the IEEE INFOCOM, 2012, pp. 2561–2565, doi:10.1109/INFCOM.2012.6195653.
[42] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, E. Porat, On finding an optimal tcam encoding scheme for packet classification, in: Proceedings of the IEEE INFOCOM, 2013, pp. 2049–2057, doi:10.1109/INFCOM.2013.6567006.
[43] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, M. Horowitz, Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn, in: Proceedings of the ACM SIGCOMM, 2013, pp. 99–110, doi:10.1145/2486001.2486011.



Shouxi Luo is currently a Ph.D student in University of Electronic Science and Technology of China, Chengdu, China. His research interests include data center networks and software-defied networking.



Hongfang Yu received her B.S. degree in Electrical Engineering in 1996 from Xidian University, her M.S. degree and Ph.D degree in Communication and Information Engineering in 1999 and 2006 from University of Electronic Science and Technology of China, respectively. From 2009 to 2010, she was a Visiting Scholar t the Department of Computer Science and Engineering, University at Buffalo (SUNY). Her research interests include network survivability and next generation Internet, cloud computing etc.

Lemin Li graduated from Jiaotong University, Shanghai,China in 1952, majoring in electrical engineering. From 1952 to 1956 he was with the Department of Electrical Communications at Jiaotong University. Since 1956 he has been with Chengdu Institute of Radio Engineering (now the University of Electronic Science and Technology of China). From August 1980 to Aug. 1982, he was a Visiting Scholar in the Dept. of Electrical Engineering and Computer Science at the University of California at San Diego, USA, doing research on digital and spread spectrum communications. His present research work is in the area of communication networks including broadband networks and wireless networks.