Fast Incremental Flow Table Aggregation in SDN

Shouxi Luo, Hongfang Yu, Le Min Li
Key Laboratory of Optical Fiber Sensing and Communications, Ministry of Education
University of Electronic Science and Technology of China, Chengdu, P. R. China, 611731
rithmns@gmail.com, {yuhf,lml}@uestc.edu.cn

***Abstract*—In OpenFlow-based SDN, flow tables are TCAM-hungry and 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 represent the same forwarding semantics. Unfortunately, the aggregation retards table updates and lengthens the updating time. 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 enough. In this paper we propose offline FFTA (Fast Flow Table Aggregation) and its online improver iFFTA to shrink the flow table size and to provide practical fast updates. iFFTA is the first online non-prefix aggregation scheme. Extensive experiments demonstrate: (1) FFTA is about $200\times$ faster than the previously published best non-prefix aggregation scheme without loss of compression ratio on offline aggregation; and (2) iFFTA achieves about $3\times$ faster than FFTA on online update incorporations with a loss of an acceptable compression ratio per update. Thus the user could make a combination use of FFTA and iFFTA for table aggregations: call iFFTA usually and recall the efficient FFTA once the switch is running out of concrete flow table space.**

I. Introduction

In OpenFlow-based SDN, forwarding tables (i.e. flow tables) are TCAM-hungry 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), and the fields continues to grow as more fields are added in [1]. Unfortunately, because of TCAMs are board-space costly, power-hungry and expensive[2]–[5], commodity OpenFlow switches suffer restricted concrete flow table space[5]–[7].

One promising direction in reducing the demands of TCAMs is flow table aggregation, a technique that merges multiple flow entries into one without modifying forwarding semantics. Moreover, because of the aggregation is a software solution, it is easy to implement it as an optional plug-in on OpenFlow controller, and such a solution does not require any changes to OpenFlow protocols or OpenFlow switches.

While a number of literatures have proposed aggregation schemes for traditional prefix IP routing tables[8]–[10] or non-prefix TCAMs rules or ACLs[2]–[4], [11], the aggregation of flow table in SDN has its own particularities.

This work was partially supported by the National Grand Fundamental Research 973 Program of China under Grant (No. 2013CB329103), and Natural Science Foundation of China grant (No. 61271171).

1. Firstly, the match fields in the flow table are non-prefix since multiple types of fields (both prefix and non-prefix) are included, and those specified prefix aggregations cannot cope with (e.g. [3], [8]–[10]).
2. Secondly, the actions of a flow table are more varied than those of ACLs, which have about 2 or 4 actions generally. So those 2 or 4-action dedicated aggregation schemes do not work well (e.g. [2], [11]).
3. Thirdly and crucially, the flow table aggregation in SDN is efficiency-sensitive especially, since forwarding rules are update-prone and the aggregation will retard table updates and lengthen the updating time. Moreover, during the update, 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[12]. So inefficient offline non-prefix aggregations are inapplicable here (e.g. [2], [4]).

To achieve practical flow table aggregation, we present a pair of aggregation schemes named FFTA (Fast Flow Table Aggregation) and iFFTA (improved-FFTA) in this paper. FFTA is an offline aggregation scheme and shares the basic 3-step aggregation framework with bit weaving[4], i.e. (1) cut the rule list into prefix-permutable partitions, then apply (2) modified prefix aggregation and (3) bit merging (merge together rules that differ by a single bit iteratively) to each partition to reduce the amount of rules. However FFTA employs an entirely different technique modified from ORTC[8] in partition aggregations and could achieve about $200\times$ acceleration without any loss of compression ratio. Based on FFTA, we further propose its online improver-iFFTA, to incorporate incremental updates efficiently.

We summarize our two main contributions as follows:

1. FFTA, a very efficient offline non-prefix aggregation scheme that is about $200\times$ faster and much more memory-efficient (orders of magnitude) than the presently best reported scheme bit weaving[4] while achieving the provable same compression ratio. For instance, for a common 100-rule partition, bit weaving costs more than 1s to aggregate while FFTA only needs several milliseconds.
2. iFFTA, an online improver of FFTA that achieves about $3\times$ acceleration further on incorporating updates with a loss of only a small quantity of compression ratio. In consideration of keeping a table most-aggregated-online is valueless in practice, the controller could make a combination use of

978-1-4799-3572-7/14/$31.00 ©2014 IEEE

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

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

Fig. 1. Two semantic equivalent toy flow tables: the left is semantic redundancy richer than the right. z is the priority, m is the match field(s), and a is the related action. The left table can be cut into two prefix-permuatable partitions, $(1, 2, 3, 4)$ and $(5, 6, 7)$.

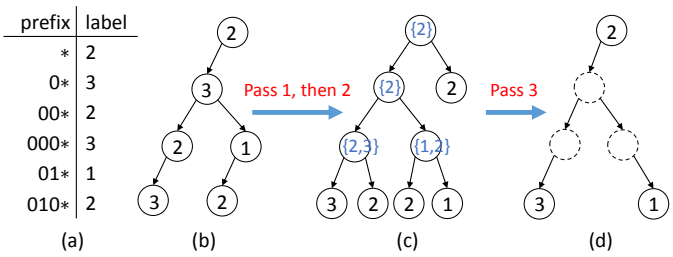


Fig. 2. An example of how ORTC works: (a) tabular form with prefix IP address in binary format and next-hop address label; (b) BST(binary search tree) with state transitions marked; (c) Pass-1 produces the leaf-pushed BST, then Pass-2 gets the set of candidate nexthops for each inner node; (d) and the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

FFTA and iFFTA for table aggregations: call iFFTA usually and recall the efficient FFTA once the switch is running out of concrete flow table space again.

The remainder of this paper is organized as follows. Section II gives an overview of flow table and the state of rule aggregations. Section III describes the design of FFTA and iFFTA. Section IV evaluates their performances. Finally, sections V and VI present related work and conclusions respectively.

II. Background and Motivation

A. SDN and Flow Table

In OpenFlow-based SDN, forwarding policies are translated into flow tables to act out[1], [13]. A flow table consists of prioritized entries, each entry may be simplified as a triple tuple $\langle m, a, z\rangle$ as the toy table in Fig.1 shows. In practical terms, the match field is the combination of ingress port, packet headers that defines the flow(such as VLAN ID, Ethernet src/dst addr, 5-tuple etc.), and optionally metadata specified by a previous table. The corresponding action a is a sub-collection of instructions that are executed when a packet matches the rule entry, including *forwarding*, *drop*, *modification*, *encapsulation*, *tunnel to controller* and etc. In the paper, we call such a match-action entry $\langle m, a, z\rangle$ a rule. We also use *rule* to denote the ternary string m (i.e. the match field) in that entry, when no ambiguity exists.

Correspondingly, a flow table with n rules can be formalized as a sequence of tuples in nonincreasing order of the priority: $\langle m_1, a_1, z_1\rangle, \cdots, \langle m_n, a_n, z_n\rangle$, where $z_1 \leq \cdots \leq z_n$ (with smaller numbers meaning higher priority in the paper). We assume they are collision-free, i.e. the rules with the same priority would not match the same packets. So, the action of a packet is exactly defined by the first matched rule unambiguously [1], expressed as $T(p)$, where p and T denote the packet and matched flow table respectively. Since forwarding rules are generated alone, multiple entries may overlap or have the same action ($a_i = a_j$ for some $i \neq j$). Aggregating those redundancy-rich rules into fewer would reduce the TCAMs demands for hardware switches or accelerate the matching for software switches. However the aggregation must not change the action of any packet (i.e. the forwarding semantics), or must preserve *semantic equivalence* so called. Suppose $T^\dagger$ is one aggregated table of T, there must be $T^\dagger(p) \equiv T(p)$ for $\forall p$. A toy example of flow table aggregation is shown in Fig.1.

[1] In OpenFlow, the packets that no rule matches with will be processed as the pre-defined/default rule specifies (e.g. drop, delivery to the next flow table or encapsulate then forward to the controller). So we assume the table to be complete here.

B. Prefix Aggregation

The aggregation of prefix rules has been widely studied more than a decade[8]–[10]. In 1998, Draves et al.[8] designed the Optimal Routing Table Constructor (ORTC) to minimize the IP routing table size, which is provably optimal by the number of rules, without altering any forwarding semantics. ORTC uses a binary search tree (BST, or prefix tree, or trie) to organize the prefix rules and employs leaf-pushing and relabel techniques to aggregate. It consists of three passes over the tree as Fig.2 shows.

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.
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.

While ORTC is an offline algorithm, several literatures (e.g. SMALTA[9] and FIFA[10]) design variants to achieve fast incremental update for aggregated rule with the sacrifice of compression ratio or recomputing time.

C. Non-Prefix Aggregation

Unlike IP routing tables, the match fields in flow table are fixed length and non-prefix (i.e. wildcards can appear at any positions in the match field). Their aggregation is conjectured to be NP-hard and several heuristics have been proposed to achieve offline aggregation[2]–[4], [11]. To the best of our knowledge, bit weaving is the best reported scheme presently, which is claimed to achieve an average compression ratio of 23.6% for general non-prefixes in their tests.

Bit weaving is based on a crucial observation that, a group of non-prefixes can be permuted into prefixes simultaneously, iff their wildcard positions are in a chain of subset relationship. Specifically, bit weaving aggregates non-prefixes by cutting them into a series of prefix-permutable partitions and making

a 4-pass aggregation over each. The partition cutting is easy to do by checking the *across pattern(s)* among continuous rules orderly and greedily [2]. But the 4-pass aggregation is quite complicated and inefficient. A brief but incomplete description of 4-pass aggregation follows:

1. **Pre-permutation:** Sort ternary bit positions in increasing order by the number of ternary strings that have a $*$ in that bit position to permute rules into prefixes.
2. **Prefix aggregation:** Create a fake default rule assigned with a fake action to make the partition complete. Assign a specific weight to the fake rule, then employ the weighted one-dimensional prefix aggregation algorithm in TCAM razor[3] to minimize the prefixes in the partition. Due to its specific weight, the fake rule always remains in the results, remove it.
3. **Bit merging:** Search and merge together rules that differ by a single bit within the partition.
4. **Post-permutation:** Revert all ternary strings back to their original bit order.

D. The Shortcoming of Bit Weaving

Bit weaving is effective but inefficient. In tests, we find that although the time of aggregating a partition in bit weaving grows linearly with the partition size, the coefficient is still too large. For instance, the costs of aggregating a 10-rule and an 100-rule partition are larger than 25ms and 1s respectively, which are quite huge delays in production networking. The statistics of [4] shows that about 2.7% of partitions have more than 32 rules and 0.6% of partitions have more than 128 rules for their real-life rules. We speculate the flow table of the further switch would have more fat partitions. Thus the delay of aggregation is considerable and bit weaving is inapplicable to dynamic networks since an inefficient global recomputing (the whole partition or even the whole flow table) is need once a rule updates.

III. Design

In this section, we describe the designs of FFTA and iFFTA which explain why they are so efficient.

A. Offline FFTA for Fast Snapshot Aggregation

FFTA (Fast Flow Table Aggregation) shares the same basic idea and algorithm framework with bit weaving, i.e. first cut the non-prefix table into prefix permutable partitions and then aggregate each partition respectively. But it employs a quite different core method in aggregating each partition. Such a method not only eliminates the two permutations (refer to Section II), visualizes the aggregation procedure, but also could accelerate the aggregation $200\times$ without loss of compression ratio, and furthermore makes the aggregation of rules traceable and easy to update.

[2] Suppose $S_*(x)$ and $S_*(y)$ are the wildcard position sets of ternary string x and y resp., x and y form a $cross\ pattern$ iff $S_*(x) \not\subseteq S_*(y) \wedge S_*(y) \not\subseteq S_*(x)$, see the Fig.1 for an example and refer to bit weaving[4] for more details.

Our novel partition aggregation is based on a key intuition: ORTC constructs prefixes into a BST (binary search tree) to do optimal aggregation simply, efficiently and visually. As the rules in each partition can be permuted into prefixes, why do not we just construct the partition to a BST-like tree then employ the ORTC-alike techniques directly? Such a technique will omit the permutations and simplify the prefix aggregation processes. Based on this, we redesign the aggregation of each partition into three steps: (I) Tree Construction, (II) ORTC-based Aggregation and (III) Bit Merging on the Tree, as the brief example in Fig.3 shows.

1) Tree Construction: Each partition here can be permuted into prefixes by sorting ternary bit positions in increasing order by the number of ternary strings that have a $*$ in that bit position (the first step for partition aggregation in bit weaving[4]). Then those artifactitious prefixes can be organized as a BST, where each artifactitious prefix is a node in it. Suppose the preimage of the lowest common ancestor (LCA) of those artifactitious prefixes in that BST is m, we can expand the wildcards that appear in m but absent from the original rules, into 0 and 1 in turn, to build up a tree to organize the partition. Obviously, such a tree is equivalent to the one constructed with artifactitious prefixes, but no permutations are needed anymore. We call it a modified-BST. Take the toy partition in Fig.3-(a) as an example, the corresponding LCA is $***1$ and its modified-BST is shown in Fig.3-(b).

WLOG, for a partition with n rules (labeled $1, \cdots, n$ resp.), its LCA (denoted as m) is calculated by $\bigvee_{i=1}^{n} m_i$, where m_i is the i-th rule' match field and the operation $\vee$ on fixed length ternary strings x and y [3] produces a new ternary string z, whose k-th bit (denoted as $z[k]$) is $*$ if $x[k] \neq y[k]$, or $x[k]$ otherwise. Then the wildcards to be expanded are those in the position set: $\bigcup_{i=1}^{n} \{k \mid m[k] \neq m_i[k]\}$.

In the paper, we built up the modified-BST recursively, as the pseudocode in Fig.4 shows. The code NODE($rule, left$=$right$=nil) denotes creating a node to store the $rule$ and letting both its left and right children (denoted as $left$ and $right$ resp.) be NULL(nil). For a node n, $n.m, n.a$ and $n.z$ denote the match field(m), action(a) and priority(z) of the stored rule respectively. In the program, we first calculate the LCA ternary string of the partition, denoted as m (Line 2 in Fig.4), and use it to create the root node to start the procedure (the root stores the fake rule $\langle m, nil, \infty \rangle$ initially). Then the construction is carried out by adding rules into the tree in nonincreasing order of the amount of wildcards in them. Each round (i.e. the APPEND-RULE in Fig.4), we push the rule to be added from the root node to leaves, until the match field sorted in the node is the same as its match field. If no node is found, we expand the last visited leaf and re-push. Finally, the rule's host node will be found or created, we update the node if the rule has a higher priority than that stored in the node.

2) ORTC-based Aggregation: After the modified-BST is built, we use ORTC to aggregate. Unfortunately, as the ex-

[3] $x \vee y$ and $y \vee x$ are nondistinctive here.

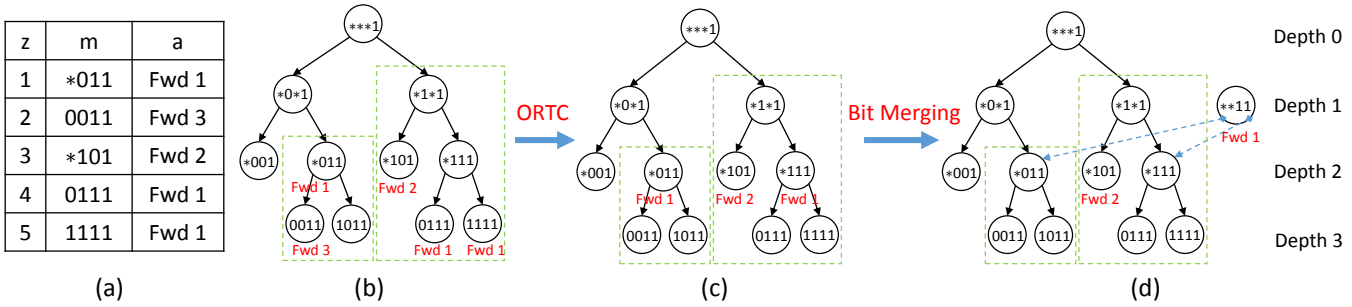


Fig. 3. An example of how FFTA aggregates a prefix-permutable partition: (a) tabular form with non-prefix match field in binary format and action; (b) modified-BST with action marked; (c) modified-BST with modified-ORTC produced; (d) the trace of bit merging and the aggregated modified-BST.

function CONSTRUCT-TREE(P) ▷ P is a list of rules.
  $m \leftarrow \bigvee_{rule \in P} rule.m$; ▷ calculate the LCA.
  $root \leftarrow$NODE($\langle m, nil, \infty \rangle, left$=$right$=$nil$);
  for each $rule$, in nonincreasing of the amount of $*$ **do**
    APPEND-RULE($root, rule$);
  end for
  return $root$;
end function

procedure APPEND-RULE($node, rule$)
  if $rule.m = node.m$ **then**
    if $rule.z < node.z$ **then**
      $node.a \leftarrow rule.a; node.z \leftarrow rule.z$;
    end if
    return
  end if
  if $node.left = nil$ **and** $node.right = nil$ **then**
    $t \leftarrow \min\{k \mid node.m[k] \neq rule.m[k]\}$;
    Expand $node.m[t]$ into $\{0, 1\}$, get $\{m^0, m^1\}$ resp.;
    $node.left \leftarrow$ NODE($\langle m^0, nil, \infty \rangle, left$=$right$=$nil$);
    $node.right \leftarrow$ NODE($\langle m^1, nil, \infty \rangle, left$=$right$=$nil$);
  end if
  if $rule.m$ belongs to $node.left.m$ **then**
    APPEND-RULE($node.left, rule$);
  else
    APPEND-RULE($node.right, rule$);
  end if
end procedure

Fig. 4. The procedure of constructing the modified-BST of partition P.

ample in Fig.3 shows, the partition may be incomplete and we can only aggregate those complete subtrees respectively. A tree/subtree is complete iff all the matched ternary strings in it have specified actions. Fortunately, it is easy to figure out the active action for each node after Pass-1 employed. However, the Pass-1 used here is a little different from the one described in Section II. This is because: (1) any interior node will cover all its descendants with the lower priority and (2) each node is either a leaf node or an interior node with exactly two children here. So, only priority-based action pushing is needed, we call it modified Pass-1 and its corresponding pseudocode is shown

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

Fig. 5. The pseudo-code of modified Pass-1 of ORTC.

in Fig.5. Now, all the maximal complete subtrees (MCS) are obvious, then Pass-2 and Pass-3 of ORTC process successively.

Such an ORTC-based partition aggregation is simple, efficient and intuitive. What's more, it shares the same optimality with the weighted one-dimensional prefix aggregation algorithm used in bit weaving[4]. The basic idea in proving their equivalence is as follows:

WLOG, suppose all the length K ternary strings in a given partition P are permuted already. Let $\{a_1, a_2, \cdots, a_n\}$ be the action set of P. Bit weaving first assigns each action a weight of 1, then creates an all-$*$ default fake rule, assigns it a fake action a_{n+1} and gives it a weight of 2^K. Since the algorithm used in bit weaving[4] outputs a prefix list whose sum of the actions' weights is the minimum, such a weight assignment guarantees that action a_{n+1} only appears in the last rule in the minimized prefix list. That is to say, no actual action will be assigned to those nodes on the paths from the root to each maximal complete subtree in the corresponding modified-BST. It is equivalent to do aggregation within each maximal complete subtree respectively. As both our ORTC-based aggregation and the weighted one-dimensional prefix aggregation do optimal aggregation to each maximal complete subtree in the partition, they have the same level of performance on compression ratio.

3) Bit Merging on the Tree: Within each partition, bit merging aggregates rules via merging two rules that have the same action and differ by a single bit into one entry by replacing the bit in that position with $*$ iteratively. Obviously, those mergeable rules contain the same amount of $*$s and they are on the same level in the modified-BST. So, we do merge from the bottom up in the tree for acceleration. At every turn,

we try to merge an unmerged rule with another (use unmerged rules preferred), and the generated rule is added into the upper level for more probable merging, as Fig.3-(d) shows. Once all mergeable rules in this level are merged (once at least), we move to the upper level and recur. The merging of rules is easy to model as a directed acyclic graph (DAG) with a vertex for each related rule and an edge for each merging operation. Moreover, since bit merging only occurs in rules with the same action, the traces of merging form disjoint DAGs for actions. We use them to accelerate the incorporation of updates in iFFTA.

Finally, we order the aggregated rules from the bottom to the top to acquire the rule orders (i.e. priorities) in the partition. To keep the relative order of rules in different partitions unchanged, we simply reuse the original priority values in that partition. It is feasible because the amount of rules would not exceed after aggregating. Although such a strategy may make some holes in priorities, it doesn't really matter. Leaving holes in priority sets or TCAM blocks is a practical trick to reduce the overhead of moving rules for adding/deleting rules.

B. Online iFFTA for Fast Incremental Update Incorporation

In FFTA, the original table is cut into partitions and each partition is aggregated independently. Thus the general approach to incorporate an update (insertion, deletion, or modification of one rule) is: (1) locate the partition where the update occurs by consulting the partition id of the pre-existing rule (for deletion or modification) or comparing priorities (for insertion); then (2) apply the update to the affected partition.

To update a partition, the naive but time-consuming way is to rerun FFTA for the entire partition. In general, most of the aggregated rules would not change for an update, therefore a more efficient approach for updates is to only re-aggregate those affected rules. Following the principle, we propose iFFTA, a suite of update incorporation strategies basing on FFTA, for fast update incorporations.

iFFTA consists two main operations: (1) update the affected nodes in the modified-BST then (2) redo bit merging for the affected rules (i.e. update the DAGs). It is easy to find the affected rules for bit merging by checking the changes of DAGs, once the updates of modified-BST are finished. Next, we present the basic principles of how to update a modified-BST in Fig.6 and give a brief introduction below.

1) Insertion: For a rule to be inserted (denoted as r), we first check its priority (i.e. $r.z$) against all the original rules (denoted as set P_o). If $r.z < \min Z$ or $r.z > \max Z$ where $Z \leftarrow \{r_i.z \mid r_i \in P_o\}$, we mark it *unaggregated* and insert it directly; Otherwise, we need to locale the rule's corresponding host node in the tree. If the host node exists, since the insertion only affects those rules in the same MCS (Maximal Complete Subtree), the insertion is analogous to an update of aggregated prefixes. We can employ SMALTA[9] or FIFA[10] for the affected MCS, or simply rerun modified-ORTC for the subtree rooted at the node (the one we implement in our evaluation). In other cases, there is no such a host node, it means the newly inserted rule introduces *cross patterns* with original rules and the modified-BST must be reconstruction [4]. We just rerun FFTA for the whole partition.

```
1:  procedure INSERT(rule)
2:      Let Z ← {r.z | r ∈ P_o};    ▷ P_o is the original rules;
3:      if rule.z > max Z or rule.z < min Z then
4:          mark rule as unaggregated, add it directly;
5:      else if the corr. host node of rule exists then;
6:          run SMALTA or FIFA for the affected MCS;
7:          rerun bit merging for the affected rules;
8:      else
9:          rerun FFTA for the partition;
10:     end if
11: end procedure
1:  procedure DELETE(rule)
2:      if rule is marked unaggregated then
3:          delete it directly;
4:      else if the deletion of rule breaks completeness then;
5:          rerun FFTA for the broken MCS;
6:      else
7:          run SMALTA or FIFA for the affected MCS;
8:          rerun bit merging for the affected rules;
9:      end if
10: end procedure
1:  procedure MODIFY(rule_old, rule_new)
2:      INSERT(rule_new);
3:      DELETE(rule_old);
4:  end procedure
```

Fig. 6. Principles in update algorithms

2) Deletion: For those rules marked *unaggregated* in insertion, we can delete them directly. Otherwise, we need to locate its corresponding host node in the modified-BST firstly. If the rule deletion does not break any completeness, e.g. one of its ancestors has a specified original action in the tree, then we rerun SMALTA or FIFA for the affected MCS. In other cases, such a deletion will break some MCS into nano MCS(s), a newly FFTA for the broken MCS is required.

3) Modification: A modification can be considered as an insertion followed by a deletion as described above.

On rare occasions when the modified-BST must be reconstruction, iFFTA degenerates into the rerun of FFTA. Fortunately, this is rarely the case, and moreover our FFTA is quite efficient.

Besides, iFFTA gives up the use of two optional techniques in bit weaving, redundancy removal (divided into upward redundancy and downward redundancy) and prefix shadowing, since they both will trigger a complex computing when any rule that makes a redundancy or shadow changes (e.g. deleted) in the original table, as we show later.

In bit weaving, a rule is called *upward redundant* iff it is

[4] Actually, inserting a rule whose match field contains the partition's LCA also causes a reconstruction even though no *cross pattern* introduced. It is avoidable by using the all-$*$ string as the partition's LCA in the initial tree construction.

completely in the shadow of prior rules and no packet will fall into it. The removal of *upward redundant* rules does not change the forwarding semantics, but make the update(s) of aggregated rules more complex. Suppose C is the set of prior rules that makes/covers *upward redundant* rules, if the match fields of any rule in C changes (e.g. deleted by controller), the aggregator has to figure out all the influenced *upward redundant* rules, then re-aggregate all the affected partitions. This may cause a global re-aggregation. Likewise, both the removal of *downward redundant* rules and the use of *prefix shadowing* of earlier partitions have the similar disadvantages.

C. About Forwarding Semantics Equivalence

Similar to the proofs in bit weaving[4] and SMALTA[9], since each step in both FFTA and iFFTA do not change the forwarding semantics of table, they keep the forwarding semantics equivalence all the time.

IV. Performance Evaluation

In this section, we evaluate the effects and efficiency of FFTA and iFFTA. Extensive experimental results demonstrate that: (1) FFTA is about $200\times$ faster than bit weaving while using much less memories and sharing the same compression ratio; (2) Based on FFTA, iFFTA is about $3\times$ faster further in incorporating updates with an acceptable loss of compression ratio. Specifically, both FFTA and iFFTA only cost several milliseconds to several ten-milliseconds for an update. While bit weaving needs several hundred-milliseconds to seconds since it has to re-aggregate the whole partition or even the whole table in an inefficient way.

A. Methodology

We implement FFTA and iFFTA in Python. Since the authors of bit weaving only releases the implementation of the one-dimensional weighted prefix minimization they used[5], which is also written in Python, we use our accelerated version of bit merging to make a complete bit weaving. All experiments were 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 used a single processor core.

Unfortunately, we have no access to non-prefix rules of real-world networks. In consideration of that all algorithms share the same way in making prefix-permutable partitions and both the offline aggregation and online update occur in a single partition alone, we use the publicly available prefix forwarding entries from Stanford University Backbone Network [14] to synthesize partitions for tests. There are 14 operational zone Cisco routers connected via 10 Ethernet switches to 2 backbone Cisco routers (named $bbra$ and $bbrb$) that in turn connect Stanford to the outside world. Fig.7 details the 16 routing tables, where the *original*, *action* and *MCS* denote the amounts of original rules, actions and maximal complete subtrees (MCS) of that prefix table with the default route (i.e. the all-$*$ entry) excluded.

[5] http://www.cse.msu.edu/%7emeinersc/suri.py

The statistics of bit weaving[4] shows that about 2.7% of partitions have more than 32 rules and 0.6% of partitions have more than 128 rules for their real-life rules. We speculate the further switch's flow tables would have more fat partitions. To make extensive experiments, we test offline aggregations on $10, 20, 30, \cdots, 300$-rule synthesized partitions, and test updates on 150-rule synthesized partitions. All the rules are selected from one of the prefix tables shown in Fig.7 and each test is repeated 20 times. We use all the 16 prefix tables for tests and the figures shown in the paper are cases of using $bbrb$'s table.

B. Results

Fig.8(a) shows that the average running time of both FFTA and bit weaving (labeled FFTA and Bitweaving resp. in Fig.8) grow linearly with the number of rules in the partition, but FFTA is about $200\times$ faster than bit weaving. For instance, FFTA costs less than 21ms to aggregate a 300-rule partition while bit weaving needs about 3.5s to aggregate a 240-rule partition. Similarly, FFTA costs much less memory than bit weaving as Fig.8(b) shows, where the two y-axis(s) denote the peak usage of physical memory and virtual memory respectively. In testing, the process of bit weaving is always put in to *Disk Sleep* state and becomes a zombie process, when aggregating those partitions whose size n is larger than 250. Thus we only test partitions whose $n \leq 240$ for bit weaving.

Fig.8(c) shows the average compression ratios (calculated by $\frac{AggregatedTableSize}{OriginalTableSize}$) of FFTA and bit weaving on those partitions. It is clear that the effects of FFTA is exactly the same with bit weaving as we prove before. We also obvious that the compression ratio here is much worse than the results claimed in bit weaving, this is due to the partitions are generated randomly. We synthesize the partitions in different strategies and find that FFTA is still about $200\times$ time faster and orders of magnitude more memory-efficient.

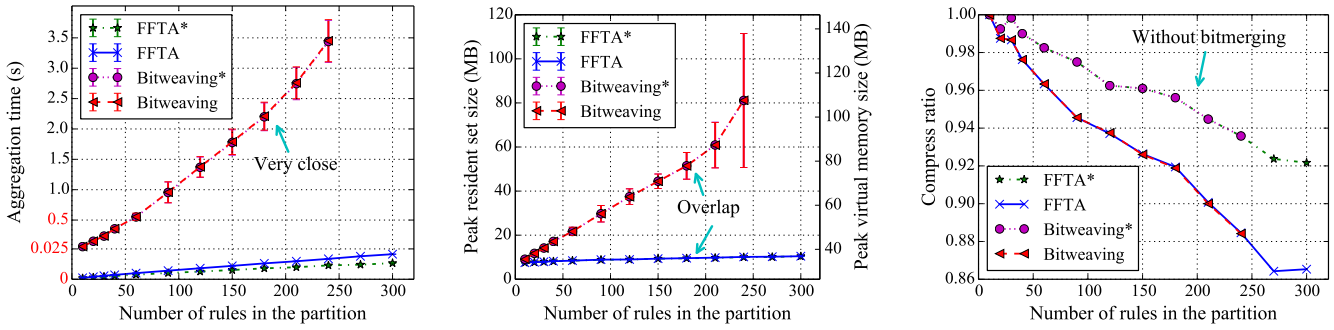
In addition, FFTA* and Bitweaving* in Fig.8 denote the results of FFTA and bit weaving with bit merging disabled respectively. We find that our accelerated bit merging does not increase the (peak) memory cost and the inefficiency of bit weaving here is mainly caused by the one-dimensional weighted prefix minimization[4], which is a dynamic programming solution.

Fig.9 shows the performances of FFTA and iFFTA on a stream of 50-rule updates. The updated partition is made up of the $bbrb$'s first 75 prefixes and last 75 prefixes. The 50 rules to be updated are randomly selected from the synthesized partition.

Fig.9(a) shows the average computing time of FFTA and iFFTA for incorporating updates. It implies that iFFTA is about $3\times$ faster than FFTA and both FFTA and iFFTA have the similar time complexity on insertion and deletion. Specifically, the time cost to incorporate an update for FFTA and iFFTA are about 3ms or 10ms respectively. What omitted in the figures, we enlarge the size of test partitions and notice that the average time per update for both FFTA and iFFTA also grow slowly with the number of rules in the partition.

router	bbra/bbrb	boza/bozb	coza/cozb	goza/gozb	poza/pozb	roza/rozb	soza/sozb	yoza/yozb
original	1825/1620	1614/1453	184909/183376	1767/1669	1489/1434	1567/1483	184682/180944	4746/2592
action	61/40	25/26	42/41	20/20	18/17	17/15	48/39	77/48
MCS	18/18	11/11	60022/60015	11/11	12/12	11/11	60015/60013	13/12
aggregated	691/662	180/156	47973/47947	147/130	103/88	97/85	47991/47956	184/115
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%

Fig. 7. The information of forwarding tables in Stanford University Backbone Network.



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

Fig. 8. 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.

We count the change of aggregated rules per update in the tests, their distributions are shown in Fig.9(b). It implies that there is very little difference between iFFTA and FFTA.

We also test the loss of compressibility for iFFTA (calculated by $\frac{iFFTA\text{-}CompressionRatio}{FFTA\text{-}CompresionRatio} - 1$). We test multiple partitions and notice that the results are tightly related to the test partition. The one shown in Fig.9(c) are the result of the case we mentioned before.

Further, we redo the experiments using all other 15 prefix tables and find the similar conclusions. Besides, the offline compression ratio of each prefix table with the default rule excluded is also shown in Fig.7.

V. Related Work

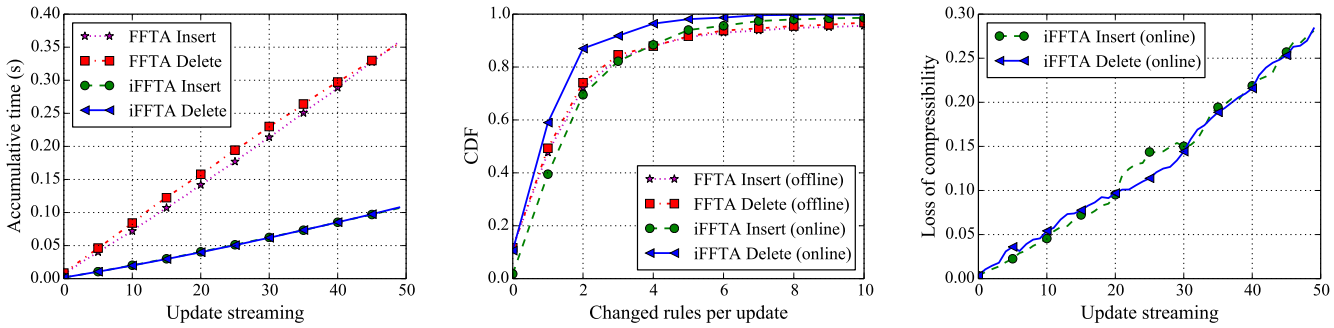
Prefix aggregation: The issue of prefix aggregation have received considerable attention from the research community over the last few years. Draves et al.[8] designed an offline algorithm called ORTC to generate the compressed IP routing table, which is proved to be optimal (means the number of entries in the generated table is minimized). Based on ORTC, online algorithms like SMALTA[9] and FIFA[10], are present to achieve the fast incremental updates of aggregated tables with a sacrifice of compression ratio or recomputing time. Although the aggregation of prefixes is different from that of a flow table, they inspire our design. In addition, we employ a variant of ORTC for partition aggregations.

Non-Prefix aggregation: The problem studied here is more similar to the aggregation of TCAM/non-prefix rules. McGeer and Yalagandula[2] formulated the TCAM rulesets minimization into a Boolean optimization problem. But their algorithms are either inefficient or customized, unpractical for flow table aggregation. Liu et al. designed TCAM razor[3] and bit weaving[4] 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 not practical for dynamic network, since global recomputing (re-aggregate a whole partition or even the whole flow table) is needed once a rule updates. Our FFTA shares the similar basic idea and achieves the same compression ratio with bit weaving, but it is more efficient (about 200× faster with less memory usage) and friendly to table updates.

More recently, Palette[6] and *One Big Switch* abstraction[7] 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 in each switch. CacheFlow[5] uses rule caching techniques to virtualize the physical TCAMs to get the illusion of an infinite rule table. While orthogonal to our work, all those works may be benefited since fewer rules would need to be distributed or cached.

VI. Conclusion and Future Work

Flow table aggregation is a promising direction in reducing the requirements of TCAMs for SDN switches. We have proposed FFTA and iFFTA, a pair of flow table aggregation and update schemes, to reduce the size of flow table with practical fast updates supplied. Extensive experiments demonstrate: (1) FFTA is about 200× faster than the best reported



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

Fig. 9. iFFTA is about 3× faster than FFTA on incorporating updates.

non-prefix aggregation scheme with much less memory usage and achieves the provable same compression ratio on offline aggregation simultaneously; (2) iFFTA achieves about 3× acceleration further with a loss of only a small quantity of compression ratio per update. For example, FFTA and iFFTA only need about 10ms and 3ms to incorporate an update respectively. So the controller could make a combination use of FFTA and iFFTA for table aggregations: call iFFTA usually and recall the efficient FFTA once the switch is running out of concrete flow table space. Since the aggregation retards table updates and lengthens the updating time, during which the networking is error-prone, FFTA and iFFTA are more practical for SDN.

The aggregation reduces the number of rules, but it also changes the definitions of those flows involved in and mixes their entries up, which results in a coarser traffic statistics. For our next step, we plan to design techniques to estimate the statistics information of each original flow from the aggregated flow.

REFERENCES

[1] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *Communications Surveys Tutorials, IEEE*, vol. 16, no. 1, pp. 493–512, First 2014.

[2] R. McGeer and P. Yalagandula, "Minimizing rulesets for tcam implementation," in *INFOCOM 2009, IEEE*, 2009, pp. 1314–1322.

[3] A. X. Liu, C. R. Meiners, and E. Torng, "Tcam razor: a systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1109/TNET.2009.2030188

[4] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: a non-prefix approach to compressing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, vol. 20, no. 2, pp. 488–500, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1109/TNET.2011.2165323

[5] N. Katta, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," Princeton University, Tech. Rep. TR-966-13, Oct 2013. [Online]. Available: http://www.cs.princeton.edu/research/techreps/TR-966-13

[6] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *INFOCOM, 2013 Proceedings IEEE*, 2013, pp. 545–549.

[7] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *Proceedings of the 9th international conference on Emerging networking experiments and technologies*, ser. CoNEXT '13, 2013. [Online]. Available: http://www.cs.princeton.edu/ jrex/papers/rule-place13.pdf

[8] R. Draves, C. King, S. Venkatachary, and B. Zill, "Constructing optimal ip routing tables," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, 1999, pp. 88–97 vol.1.

[9] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis, "Smalta: practical and near-optimal fib aggregation," in *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '11. New York, NY, USA: ACM, 2011, pp. 29:1–29:12. [Online]. Available: http://doi.acm.org/10.1145/2079296.2079325

[10] Y. Liu, B. Zhang, and L. Wang, "Fifa: Fast incremental fib aggregation," in *INFOCOM, 2013 Proceedings IEEE*, 2013, pp. 1–9.

[11] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang, "Compressing rectilinear pictures and minimizing access control lists," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1066–1075. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283498

[12] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 323–334. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342427

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[14] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 99–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482638