RuleTailor: Optimizing Flow Table Updates in OpenFlow Switches with Rule Transformations

Bohan Zhao, Jin Zhao, *Member, IEEE*, Xin Wang, *Member, IEEE*, and Tilman Wolf, *Senior Member, IEEE*

***Abstract*—Software-defined networking (SDN) provides flexible network control, which has enabled new, more complex network control mechanisms. These approaches impose high demands on flow updates in OpenFlow switches. Existing SDN controllers and firmware have been optimized for high-speed network updates, but specific switch implementations differ in their behavior. Previous optimization schemes for updates ignore the diversity of instruction types and switch behavior. In this paper, we present *RuleTailor*, an efficient, measurement-based optimization framework for SDN flow updates, to overcome these limitations. In contrast to other measurement-based optimization frameworks, *RuleTailor* uses new techniques, such as instruction type transformation, pseudo deletion, and match field distance, which can work in both the control plane and the data plane. Our evaluation shows that *RuleTailor* achieves a performance of fewer than 12 milliseconds per update in a content-addressable flow table with 1k entries, outperforming the state-of-the-art measurement-based framework, Tango, by a factor of 10.**

***Index Terms*—Software-defined networks, flow update, rule transformation.**

I. Introduction

SOFTWARE-DEFINED networking (SDN) [1] controls network state and configurations dynamically through controllers that manage the flow-specific forwarding operations in networking switches. The finer level of control provided by SDN will lead to wide adoption of this technology in a variety of network scenarios. The response time of SDN to changes in network state is a key performance metric for such systems. For example, delay constraints for network reconfiguration [2] and topology change [3] have been stated as 25 milliseconds or less. Similarly, failure recovery [4] is highly time-sensitive to guarantee network stability and avoid large-scale packet loss.

The response time of an SDN system to updates depends on several sequential steps: rule generation by the controller, instruction transmission to switches, switch execution, and confirmation. The delay from the third step (switch execution), which we call flow update time, has become a main bottleneck in the SDN response loop. According to the recent measurements of flow table consistency [5], there is a big gap between the control plane confirmation time and when data plane probes are affected (up to 3 milliseconds depending on flow table occupancy), which is a much slower rate than rules distribution in the control plane. Thus, the flow update performance presents a critical bottleneck in SDN. Flow update delay mainly comes from hardware writing cost. Current off-the-shelf switches typically use ternary content-addressable memory(TCAM) to exploit hardware parallelism during flow table lookups. Due to the existence of wildcard rules, one packet may match multiple flow table entries at the same time. As only the rule with the highest physical address in TCAM can be returned, it may potentially cause a large number of entry movements in the TCAM to preserve correctness in entry order. These movements are the primary source of the hardware delay for flow updates, which significantly grows with TCAM occupancy. Reducing flow update delay is an important challenge and our paper aims to address this challenge.

Several solutions have been proposed to reduce this hardware delay for flow updates. In the control plane, the number of flows can be minimized or their structure can be optimized [6–8]. In the data plane, changes to firmware have been proposed: RuleTris [9] and FastRule [10] change the update mechanism on physical switches to avoid unnecessary flow entry movements using Directed Acyclic Graph (DAG). Shah and Gupta's algorithm [11] changes the storage structure of TCAM by keeping free spaces in several parts of memory. Measurements have shown the performance of today's flow updates ranging from 30 to 100 milliseconds [5, 12, 13].

However, there is another critical factor that determines the performance of flow updates: switch implementation. A broad diversity of switch implementations exists, which encompasses software control and hardware properties, influences the efficiency of update operations. Most prior research on SDN flow updates has assumed that all switches have the same hardware and software behavior. Ignoring these differences can lead to substantial performance challenges. For example, FLIP [7] searches for an optimal rule distribution in a global network. But since different switches perform differently when executing different types of operations, FLIP in practice cannot be as effective as assumed(e.g., if addition costs much more time in switch A than in switch B, inserting two rules in A and one in B may not be faster than inserting one in A and three in B). Tango [14] has noticed this problem and proposes executing operations in priority order according to measurement results. However, Tango considers modification and deletion times as constant and leaves out operation transformations. Thus, Tango suffers from a single operation bottleneck.

To overcome the bottleneck of flow update efficiency in

Manuscript received April 30, 2019; revised June 17, 2019; accepted October 6, 2019. The work was supported in part by Natural Science Foundation of China under Grant 6197210. (Corresponding author: Jin Zhao.)
B. Zhao, J. Zhao, and X. Wang are with the School of Computer Science, Fudan University, Shanghai 200433, China, and also with Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 200433, China. (e-mail: bhzhao16@fudan.edu.cn; jzhao@fudan.edu.cn; xinw@fudan.edu.cn)
T. Wolf is with the Department of Electrical and Computer Engineering, University of Massachusetts Amherst, MA 01003, USA. (e-mail: wolf@umass.edu)

consideration of switch heterogeneity, we propose RuleTailor, an efficient measurement-based optimization framework for general TCAM-based SDN switch flow update. In RuleTailor we propose two schemes to reduce flow update latency in both the control and the data plane, and achieves a delay of less than 12 milliseconds per update in a flow table with 1k entries. In general, our optimization framework addresses the challenge of low-latency flow table updates through the following contributions:

1) We present an algorithm in the control plane to reduce flow update delay. The algorithm preprocesses incoming update instructions in batches by detecting instruction dependencies, performing priority sorting and type transformation. The algorithm can be adapted to different commercial off-the-shelf switches based on the measurement-first approach.
2) We propose a scheme to speed up flow update in the data plane by pseudo-deletion and minimizing match field distance for modification. We also develop a special data structure called Minimum Distance Deletion Cache and the corresponding management algorithms to implement this scheme.

We have implemented RuleTailor as a standalone application and use input instructions to simulate the flow update process. Through evaluation, our solution reveals to be 10× faster than Tango [14]. Furthermore, we demonstrate that our solution is scalable through a large-scale emulation of flow tables.

The remainder of this paper is organized as follows: We first provide the background of flow updates and the OpenFlow switch architecture in Section II. Then, Section III motivates our work through measurement-based exploration of the diversity of switch-specific flow updates. In Section IV, we describe our RuleTailor framework in detail, including optimization algorithms and corresponding data structures. Then, we provide an evaluation of RuleTailor for various flow update scenarios and an analysis of the results in Section V. Section VI discusses related work. We conclude this paper in Section VII.

II. Background

In this section, we discuss the problem space for our work. We first refer to Table I for the definition of the different notations used along this paper.

A. Flow Update in Switches

Existing switches provide packet lookup by matching the packet header against the stored rules in flow table to determine the action for the packet. TCAM, a typical hardware implementation of flow table, can perform fast lookup by searching *all* the stored rules in parallel. However, due to the existence of wildcard rules, a packet may match multiple rules. To avoid ambiguity, TCAMs are designed to return the rule with the highest physical address. Therefore, the rule order in TCAM is important to preserving the correct semantics of forwarding policies.

TABLE I
Notations

Notation	Description
r	A rule in TCAM
R	Set of rules
i	A update instruction from the controller to the switch
I	Set of update instructions
IS	Sequence of instruction set
$i_1 \rightarrow i_2$	i_1 depends on i_2
F_i^j	ith match field of rule r_j
M_j	Match field set of rule r_j
$MFD(r_i, r_j)$	Match field distance between r_i andr_j
P	Set of rules affected by an instruction set
l	link element in P standing for a pair of target and final rules of the same priority
S	Universal set of IS from a specific instruction set
$R_0 \xrightarrow{I} R_t$	change ruleset R_0 to R_t by executing I
T	Execution time of a single instruction or an instruction set
$N_{i,IS}$	the number of rules whose priority is greater than i after executing all instructions ahead of i in IS
$i_1 \prec_{IS} i_2$	i_1 is ahead of i_2 in IS
$n << X$	bitwise left shift to the number n by X bits

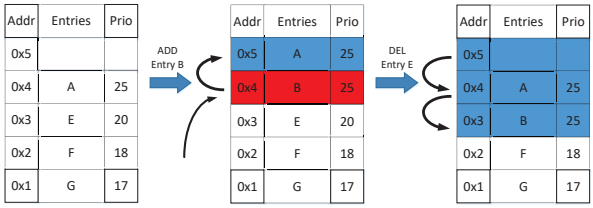


Fig. 1. Naïve insertion and deletion operations in TCAM. The blue entries are those modified during once operation. The red entry stores newly inserted rule.

When updating rules in a TCAM, maintaining the rule priority dependency may lead to time-consuming operations. There are typically three types of rule operations that can be executed in switches: addition, modification, and deletion [15]. Assuming that all rules in TCAM are regarded as a sequence according to their priorities. When inserting a rule into the sequence, all the rules whose priority is higher than the new rule need to be moved upward by one entry to create a space for this new rule. Similarly, after deleting a rule, all the rules with higher priorities need to be moved downward. The principle of this naïve flow update in a TCAM is shown in Fig. 1.

Thus, the flow update in a TCAM is similar to insertion sort algorithm that needs on average $n/2$ movements to insert a new item in a TCAM with n entries [16]. As to move a TCAM entry costs a constant amount of time (0.6ms) [9], the time complexity is too high even for a 1k-size TCAM.

Existing switches generally support the three operations by executing corresponding instructions from the controller. The actual implementation algorithms of update operations may be much more complex in real systems. To evaluate the flow

Rules	Match Field	Prio
r_1	**00	0
r_2	00**	1
r_3	01**	1
r_4	011*	2

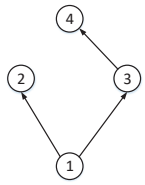


Fig. 2. Dependency relationships and minimum set of priorities of a ruleset. Here we use the directed lines from A to B to indicate that when a packet matches rule A and rule B, then B is the preferred match (i.e., A "depends on" B).

update performance of switches, we define the instruction execution time as the time between sending the controller request and time the target rule is updated, which can be affected by several factors, such as priority order and instruction type.

B. Flow Dependency

As mentioned above, TCAMs are designed for fast packet matching (i.e., search) rather than efficient updates. The reason for the slow update performance is that TCAMs must maintain the order of rules in the flow table according to priority in order to satisfy “flow dependency" [17].

When forwarding packets of the globally deployed Internet Protocol (IP), two or more forwarding rules may have the same IP address prefix. In such a case, the longer prefix (i.e., the more specific rule) should be the match result (also called “longest prefix match"). Similarly, in OpenFlow switches, each rule in the flow table has several match fields. When an incoming packet’s header matches the fields of two or more rules, then there must be an order to tell the switch which rule should be the correct match for this packet.

To state these dependency relationships, we assign a priority to each rule. Maple [18] provides an algorithm to generate the minimum set of priorities for installing rules that satisfy dependency constraints. Fig. 2 shows an example of dependency relationships and the corresponding minimum set of priorities in a ruleset. In our work, we assume that the priorities of rules are given and the rule of higher priority needs to be matched first.

C. Instruction Dependency

In network update there are two kinds of rule installations: proactive installation and reactive installation. Proactive installation allows installing rules in advance before packets arrive. While reactive installation is used in a scenario that there is no rule can be matched with the packet, so the controller installs a new rule into the switch. These two scenarios differ when we change the execution sequence of instructions, which is denoted as IS in this paper.

Sometimes we hope to process(sort or group) IS before executing it to optimize update. To guarantee the correctness of the process, there are two conditions need to satisfy:

1) Correctness: executing the processed IS' or original IS should not differs in the final semantics of the flow table. To detect and eliminate the potential conflicts during processing, we define the dependency of instructions: if instruction i_1 modifies rule r or inserts r into the flow table, and rule r is deleted or modified by instruction i_2, then i_2 depends on i_1, which is denoted as $i_2 \rightarrow i_1$. The execution order of instructions with dependency relationships cannot be reversed. In proactive installation, there are no such dependencies as all instructions aim to install rules or change existing rules in the flow table. While in reactive installation, a newly inserted rule can be changed before long because of flow change.
2) Performance: The optimizing ratio usually grows with the number of instructions we can process in batches. However, if we save instructions in batches instead of executing them at once, there is an inconsistency between the data plane and control plane, which may cause forwarding error. Therefore, we should make a trade-off between efficiency and inconsistency duration.

D. Match Field Distance

In OpenFlow switches, rule match fields describe the specific header fields of packets to match. Also, match fields are the basis for instructions to identify flow table entries. The number of match fields varies in different OpenFlow protocols. For example, there are 12 match fields in OpenFlow v1.0 [1]. This variation can be a potential factor affecting flow update performance: Suppose that there are two rules r_1 and r_2 in a flow table. We use element sets $M_1 = \{F_1^1, F_2^1, \ldots, F_n^1\}$ and $M_2 = \{F_1^2, F_2^2, \ldots, F_n^2\}$ to indicate the match fields of two rules (n is the number of match fields). Then, we define the match field distance (MFD):

$$MFD(r_1, r_2) = n - \sum_{i=1}^{n} F_i^1 \& F_i^2, \tag{1}$$

where & is the bitwise logical AND operation.

For strict modification (only the flow table entries with the same match fields and priority can be matched, and the wildcard (*) is considered as a unique element of the field) the MFD between the target rule and the modified rule can influence the delay of update. In Section III, we execute modification instructions in hardware switches with different numbers of match fields. Experimental results indicate that the MFD between the target rule and the modified rule has a significant positive correlation with the modification time. The reason may be that since OpenFlow v1.3 flow tables are implemented by group tables. In the switch pipeline, a packet is matched and forwarded by tables one-by-one, each only matching specific match fields separately. Therefore, to update a rule in a flow table, the switch has to update information in more than one hardware table.

E. Experimental environment

1) Switches Under Test: As the foundation for our work, we evaluate the performance of four hardware OpenFlow switches. The exact models and their specifications are presented in Table II.

TABLE II
SWITCH MODELS, FIRMWARE, AND OPENFLOW VERSION UNDER TEST

Switch Model	Firmware	OpenFlow
HP 2920	WB.16.02.0021	1.3
Centec V350	CentecOS3.2.4	1.0
EdgeCore AS5710	EdgeCOS 1.2.155	1.3
H3C S5130	Comware7.1.045	1.3

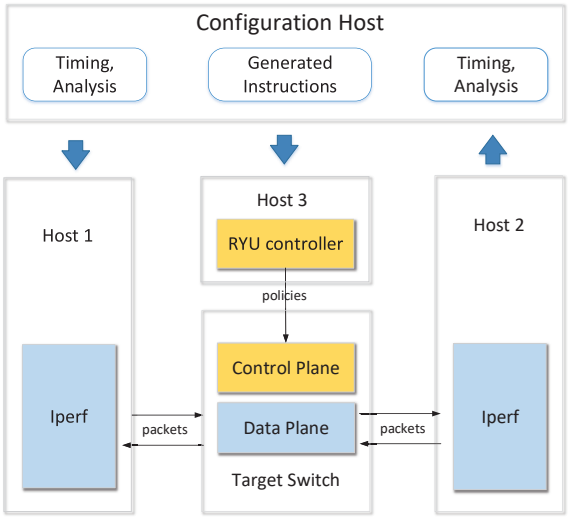


Fig. 3. Experimental testbed.

2) Experiment Testbed: Our experimental environment contains a commodity switch, a configuration host, and three servers (one as OpenFlow controller and two as end-hosts). Hosts are connected to the switch with physical cables. The controller and the switch are connected by a separate OpenFlow VLAN so that the OpenFlow network can be isolated from the experiment control network. The experiment controller connects to all other devices via VLAN. The topology of our testbed is presented in Fig. 3.

III. MEASUREMENT FRAMEWORK

A. Flow Update Time

Due to the differences in implementation of commercial off-the-shelf switches, the flow update performance varies, even for the same instructions and priority order. In this experiment, we design a procedure to measure the performance of flow updates.

We execute instructions on the same flow table. The number of instruction in each experiment is 500. To investigate the influence of priority orders and instruction types on flow update, we add, modify, and delete rules in ascending, descending, and same orders (which means all rules have the same priority). First, we clear the flow table and insert the first valid rule so that Server 2 can connect to Server 1. When measuring the insertion time, we insert 500 invalid rule in the flow table and record the beginning of insertion as T1. Then, insert the second valid rule so that Server 1 and Server 2 are bidirectionally connected. T2 is the time that Server 2 receives the first packet sent from Server 1, which is considered as the completion of the insertion. We can approximately consider T2 – T1 as the instruction execution time.

When measuring the modification time, the second valid rule is first deleted so that Servers 1 and 2 are not connected. Then, we send 500 modification instructions on an IPv4 match field from the controller to our switch. T3 is the time when the first instruction is sent to the switch. Finally, the second valid rule is inserted again. We record as T4 the time when Server 2 receives the first data packet sent by Server 1. T4 – T3 is the resulting modification time. When measuring the deletion time, we delete the rules in the flow table one-by-one in sequence. The measurement procedure is the same as for modifications.

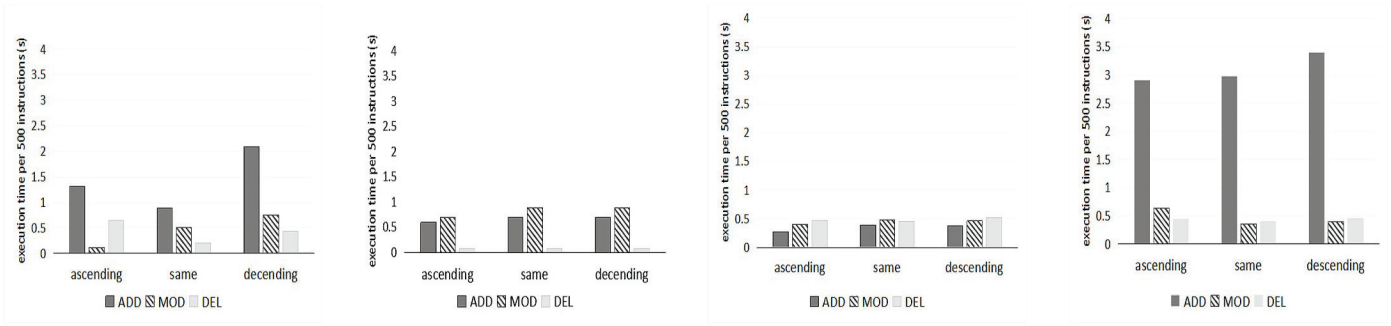
The total instruction execution times of the four switches are shown in Fig. 4. Each experiment is executed 10 times to get an average result. The ordinate value indicates the time taken for updating 500 flow entries, in seconds. We can observe that for the HP switch it is faster to execute instructions in ascending order and addition takes more time than the other two operations. In contrast, in the Centec switch, modifications take longer than addition. The reason may be that the modified flow entry needs to be restored or reordered. The deletion time is the shortest, which offers an opportunity for optimization. For the EdgeCore switch, modification and deletion are both slower than addition, and the switch performs better in ascending order. For the H3C switch, inserting rules is really slow, the reason of which may be that the implementation mechanism of addition is complicated, or the flow table size is very large (65535).

From the experimental results above, we can identify several factors influencing the instruction execution time. First, the priority order of the instruction set accounts for differences in performance. Executing instructions of the same priority takes the least time as there is no need to move rules in TCAM. The performance of ascending and descending order depends on the update algorithm. For example, we can observe that in Fig. 4 that addition in same order takes 32% less time than in ascending order, and 57% less time than in descending order. Second, the type of instructions affects the execution time. As we have seen, addition, modification, and deletion are, respectively, the slowest operation for HP 2920, Centec V350, and EdgeCore AS5710. These observations are significant as they indicate the possibility of improvement in the flow update.

B. Match Field Distance and Update Time

Similar to the measurement procedure of flow update, we execute an experiment to investigate how MFD influences flow update performance. In this experiment, we insert the first valid rule and then 500 invalid rules. Then, 500 modification instructions, which change specific match fields, are executed. Finally, a second valid rule is inserted to record the end of modification. We increase the number of match fields in each modification gradually. The modified match fields and the corresponding results are shown in Fig. 5.

We can observe that the more fields are changed in one modification (which means the greater MFD), the longer time an operation needs to be finished. Therefore, it is significant to reduce MFD when modifying rules.



(a) HP 2920 (b) Centec V350 (c) EdgeCore AS5710 (d) H3C S5130

Fig. 4. Flow update time in commercial off-the-shelf switches.

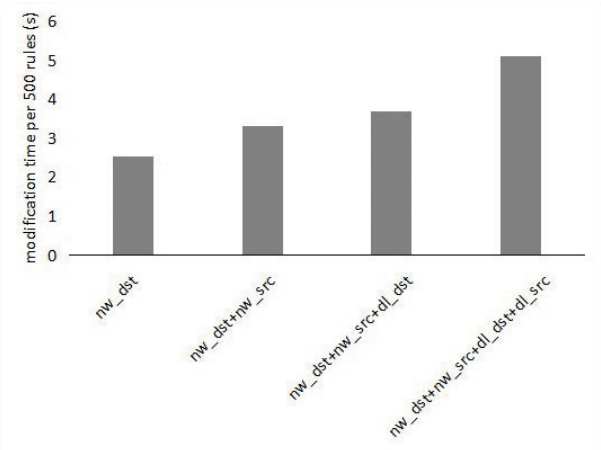


Fig. 5. Modification time for different match field distances.

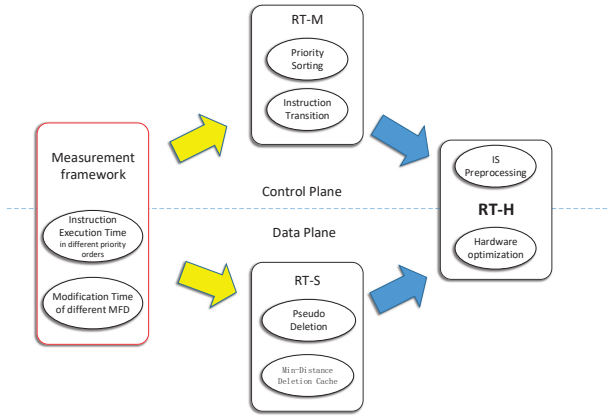


Fig. 6. RuleTailor framework overview.

C. Measurement Summary

The measurement results help us obtain a better understanding of diverse switch operation. From the experimental data, we extract three factors that influence the flow update time:

- Priority order of *IS*,
- Instruction type, and
- MFD in modification.

These observations are the foundations of our design of the RuleTailor framework. Our scheme addresses two challenges: (1) how to choose the best priority order and optimize instruction types in the *IS* and (2) how to minimize the MFD in modifications and speed up other operations. In the following section, we introduce RuleTailor and how it accomplishes these goals.

IV. RuleTailor Architecture

In this section, we introduce RuleTailor, our optimization framework for flow updates. We show the architecture of RuleTailor in Fig. 6. There are three optimization patterns in RuleTailor, which work both in the control plane and the data plane. The first pattern, named RuleTailor for Multiple instructions optimization (RT-M), works only in the control plane. RT-M takes a batch of instructions as the *IS* and then preprocesses them by sorting, integration, and transformation. The second part, RuleTailor for Single instruction optimization (RT-S), is designed to work in the switch's data plane. RT-S improves the flow update operations of switches by changing the instruction execution scheme. The last pattern, RuleTailor for Hybrid planes optimization (RT-H), is a hybrid version of the other two patterns, which applies the ideas from both and thus operates in both planes. All three optimization patterns are based on our measurement results and take operation performance of specific switches into consideration.

A. Multiple Instruction Optimization

Our first algorithm focuses on optimization at the multi-instruction level. Most production SDNs are generally based on commercial off-the-shelf switches. Therefore, optimizations that can be implemented in the control plane are more practical and flexible. This requirement motivates us to design RT-M, a measurement-based flow update optimization pattern.

1) Architecture Overview: Fig. 7 shows the architecture and operation of RT-M. First, with the help of assistant hosts, the measurement engine completes probing the target switch. The results are stored in the RT-M database as an input of the algorithm. Then, the controller interaction part sets up a batch to collect instructions. After that, the key component, RT-M Core, can utilize the measurement results to preprocess the instructions by instruction integration, type transformation,

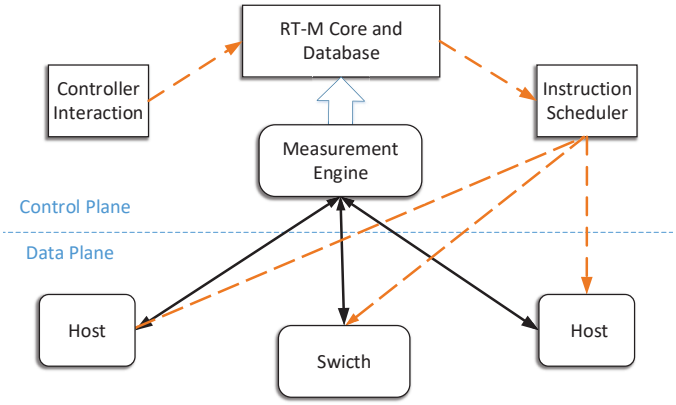


Fig. 7. RT-M architecture components. The orange lines represent the instruction flows and black lines represents data flows.

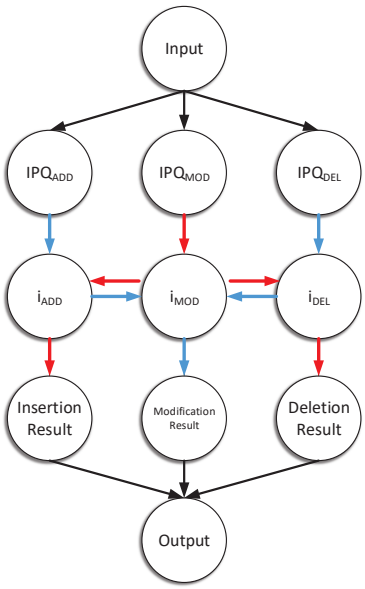


Fig. 8. Work flow of RT-M for two different scenarios. The blue lines represent workflow in the situation that the summation of the delay of an addition and deletion operation is greater than that of the modification operation, where we combine two instruction from an indexed priority queue (IPQ) to generate a new modification instruction. And the red lines represent workflow on the contrary, where we split the modification instruction into an insertion and a deletion instruction.

and priority sorting. Finally, Instruction Scheduler sends the processed *IS* to the switch, which ends the RT-M work flow.

2) RT-M Algorithm: In this section, we present the details of the preprocessing step in our RT-M algorithm. The implementation of RT-M needs to address three main challenges:

Dependency detection: In Section II, we have mentioned that to guarantee forwarding correctness during preprocessing, we should filter instructions with dependency relationships out of the batch and execute them separately (or spread them among different batches). Therefore, instruction dependency detection is necessary for our multiple-instructions optimization. Here, we denote the object of an instruction as the target rule and the outcome of an instruction as the final rule. For insertion instructions, there is only a target rule, and for deletion only a final rule exists. Modification has both rules as it changes the former to the latter. In the target *IS*, we record the final rules (if they exist) using an efficient symbol table, such as a Bloom filter. If the target rule of an instruction is the same as an instruction in the record (which means this instruction depends on the former in the *IS*), the instruction should be removed from the batch and executed later.

Order selection: Observing the measurement results in Section III, we see that instruction execution time differs by type and priority order. Thus, before further analyzing the rule transformation, we should choose execution order first. In the RT-M algorithm, we determine the best priority order (ascending/descending) for each operation according to measurement results: the order with less execution time of this operation is selected.

Transformation and integration: Having determined the best priority order for each operation, we utilize the execution time in the specific order to guide our optimization. In particular, when the summation of the delay of one insertion and one deletion is greater than that of one modification, we can combine an addition and deletion instruction of the same priority into a modification instruction. The target rule of the new instruction is the same as the deletion instruction's and the final rule is the same as the addition's. On the contrary, when the summation of the delay is less than that of a modification, we can divide the modification instruction into an insertion instruction and a deletion instruction accordingly. Since the execution time varies with TCAM occupancy, we use the average time of 0%-100% occupancy to estimate the execution time of a single switch.

Complete algorithm: Algorithm 1 shows the complete RT-M algorithm. We use Fig. 8, to illustrate RT-M operation in two different scenarios. When we output the results we first return deletion instructions, then modification and insertion at last. For each type the instructions are sorted in the selected priority order. This keeps the flow table in the smallest scale and reduces the update cost.

To sort instructions of the same type by priority and access any one of the specific priority randomly, we use a modified IPQ to store instructions. The original IPQ supports sorting elements in $O(n \log n)$ and access the key of a specified index in $O(1)$. To map the priority to these indices, we combine the IPQ with a hash tree (HT). The element in an IPQ is a bucket containing instructions of the same priority. In an HT, the key is the priority of the target bucket and the value is the index of buckets in the IPQ. When we need to push and pop instructions, we get the index according to the priority from the HT and then access the corresponding bucket in the IPQ. Finally, when outputting the results in priority order, we pop elements in the IPQ one-by-one. Fig. 9 shows an example of the function and structure of our modified IPQ.

3) Correctness Proof: Before we prove the correctness of our algorithm, we first describe the flow update problem from a more abstract perspective. We use R_0 and R_t to indicate the target and final rule set of an instruction set I and a bigger set $P = R_0 \cup R_t$ includes all rules affected during update. All

Algorithm 1: RuleTailor-M

Input: the target IS of size n, execution time in specific priority order: ADD, MOD, DEL
Result: three IPQs of different operation types

$Pattern \leftarrow \{pattern\ of\ best\ order\ for\ each\ operation$:
  $[pattern1 : ASC\_DEL + ASC\_MOD + ASC\_ADD]$
  $[pattern2 : ASC\_DEL + ASC\_MOD + DSC\_ADD]$ ...}
for $i \in IS$ **do**
  if $ADD + DEL > MOD$ **then**
    if $i.type = ADD\ or\ DEL$ **then**
      $pop\ j\ from\ another\ IPQ$
    end
    $i_{MOD} \leftarrow i + j$
    $IPQ_{MOD} \leftarrow i_{MOD}$
  end
  if $ADD + DEL < MOD$ **then**
    if $i.type = MOD$ **then**
      $divide\ i\ into\ i_{ADD}\ and\ i_{DEL}$
      $IPQ_{ADD} \leftarrow i_{ADD}$
      $IPQ_{DEL} \leftarrow i_{DEL}$
    end
  end
  if $i\ not\ processed$ **then**
    $IPQ_{i.type} \leftarrow i$
  end
end
return IPQ_{DEL}, IPQ_{MOD}, IPQ_{ADD}

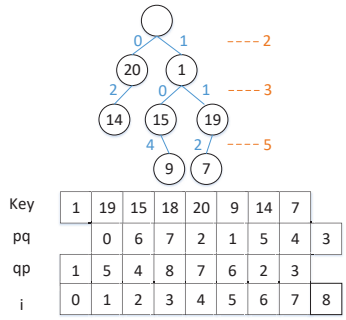


Fig. 9. An example of storing instruction buckets of different priorities in our modified indexed priority queue.The Key array stores the bucket of different priorities. The pq array stores the index of the key in IPQ position i. The qp array stores the IPQ position of the key with index i. The HT nodes marked by priority store the IPQ positions of buckets. If we want to access the bucket of priority 15, we access node 15 in the HT to get the index of the bucket (2), and then we access $Key[2]$ to get the target bucket.

possible IS from I that can finish the update make up the set S_I. If there is a rule $r_0 \in R_0$, $r_t \in R_t$, and r_0, r_t have the same priority, we use a link l_0 to replace them in the set P. We use ADD, MOD and DEL to represent three kinds of operations in update problem. Then, we have Lemma 1.

Lemma 1: $\forall I \in S : R_0 \xrightarrow{I} R_t,\ s.t.\ |I| \geq |P|$

Lemma 1 tells us that we cannot find such an IS that is smaller than P, which means that for each element in P at least one instruction is needed to implement the update.

Proof: As mentioned above, there are three kinds of elements in P: r_0, r_t, l. For each element at least an instruction is needed, otherwise it could not be affected during update. MOD can affect more than one rules but in P as we link all rule of the same priority in pair, it can only affect one l elements, too. l element could also be affected by an ADD and DEL, so $|I| > |P|$ sometimes. ■

From Lemma 1, we can get the following corollary.

***Corollary** 1:* $Card\{i \mid i.type = MOD, i \in I\} \leq Card\{l \mid l \in P\}$

Proof: Lemma 1 tells us that each l in P could be affected by a MOD or an ADD and DEL. Since l is the only type of elements in P that can correspond to modification, it accounts for all instructions of this kind. In other word, if we use only MOD to deal with ls, we get an I of minimum size: $|I| = |P|$. Therefore we get the following conclusion: $|I|-|P| = Card\{l \mid l \in P\} - Card\{i \mid i.type = MOD, i \in I\}$, which proves our corollary. ■

Our algorithm is proved by the following theorems.

Theorem 1: If $T_{MOD} < T_{ADD} + T_{DEL}$ and $T_{I_0} = Min\{T_I\}$, then $Card\{i \mid i.type = MOD, i \in I_0\} = Card\{l \mid l \in P\}$.

Here, we use T to indicate the execution time of an operation or an instruction set I. Without loss of generality, we suppose MOD operations take less time than the sum of other two operations. According to the proof of Corollary 1, we have another expression of this theorem: If $T_{MOD} < T_{ADD} + T_{DEL}$ and $T_{I_0} = Min\{T_I\}$, then $|I_0| = |P|$.

Proof: Corollary 1 has shown that the number of MOD instructions is no greater than l elements in P. We suppose that there is another I' of the same semantics as I_0. $T_{I'} = Min\{T_I\}$, and $Card\{i \mid i.type = MOD, i \in I'\} < Card\{l \mid l \in P\}$. Thus, $\exists l_0 \in P$, l_0 is affected by an ADD and DEL. Since $T_{MOD} < T_{ADD}+T_{DEL}$, if we use a MOD instruction to replace the ADD and DEL instructions corresponding to l_0, we get a new instruction set denoted as I'', s.t. $T_{I''} < T_{I'}$. This is contrary to our hypothesis. Therefore, our Theorem 1 has been proved. ■

Theorem 2: $\forall IS \in S_I$, if we move all ADD instructions to the head of IS to get IS_{sup}, and move all ADD to the end of IS to get IS_{inf}, then $T_{IS_{inf}} \leq T_{IS} \leq T_{IS_{sup}}$ (if the relative positions of same-type instructions is fixed).

Theorem 2 provides us with a lower and upper bound of a specific IS's execution time, which varies with the type order. This explains the reason why we sort instructions by type in our algorithm. Here we do not take into consideration the inner order of same-type instructions as it is fixed by our measurement results.

Proof: Since the usual MOD operation can neither change nor be affected by . $\forall i_A, i_D \in IS,\ i_A.type = ADD,\ i_D.type = DEL$, we have the following conclusions:

$$
\begin{aligned}
N_{i_A,IS} &= N_{i_A,IS_{sup}} - Card\{i_D \mid i_D \prec_{IS} i_A, i_D.pri > i_A.pri\} \\
N_{i_A,IS} &= N_{i_A,IS_{inf}} + Card\{i_D \mid i_A \prec_{IS} i_A, i_D.pri > i_A.pri\} \\
N_{i_D,IS} &\leq N_{s,IS_{sup}} - Card\{i_A \mid i_A \prec_{IS} i_D, i_D.pri < i_A.pri\} \\
N_{i_D,IS} &\geq N_{s,IS_{inf}} + Card\{i_A \mid i_D \prec_{IS} i_D, i_D.pri < i_A.pri\}
\end{aligned}
\tag{2}
$$

Here, $i_A.pri$ indicates the priority of instruction i_A. Since for DEL rules of the same priority but lower address also need to

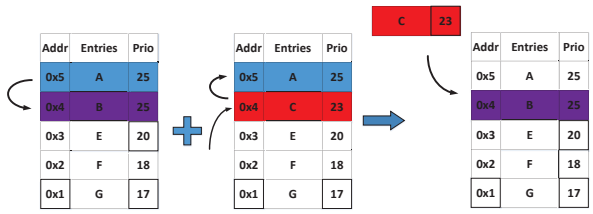


Fig. 10. Principle of pseudo deletion. The blue entries are those modified during once operation. The red entry stores newly inserted rule. The rule in purple entry is deleted.

be moved, $N_{s,IS}$ does not exactly equal the entry movements during deletion. In summary, since $N_{i,IS}$ indirectly reflects the execution time of DEL and ADD operations, combining the above conclusions, we can see that $T_{IS_{inf}}$ and $T_{IS_{sup}}$ are the lower and upper bounds of I's execution time, respectively. ■

B. Single Instruction Optimization

Having introduced the optimization algorithm in the control plane above, we turn to the data plane. If we are permitted to adjust the flow update operations to specific switch models, we can gain additional performance improvements. To apply our measurement results to single-instruction optimization, we propose RT-S to achieve efficient TCAM updates. RT-S speeds up single-instruction execution without batches, and thus improves the overall performance of switches.

1) Pseudo Deletion: Before explaining our algorithm, we first present a new concept in switch operation. When the switch receives an instruction to delete a flow entry, there is no need to delete the rule from the flow table. Instead, we can mark the entry with a mask so that it no longer participates in later matches. We define this operation as pseudo deletion. In some commercial off-the-shelf switches, a similar masking technology has been implemented to simplify the deletion operation [9], but we redefine the concept and propose a new usage.

As mentioned in Section II, the main delay of addition operation results from a large number of moves of entries. Therefore, if there is extra space for a switch to write the inserted rule directly, no movements happen. Instead, only one modification is needed to finish the insertion. Pseudo deletion provides such extra space. Fig. 10 shows the principle of pseudo deletion.

However, to guarantee the correctness of insertion, we need to make sure that rules around the extra space have the same priority as the inserted rule does. Additionally, we have to record the positions of the extra space in the TCAM. Therefore, a simple and efficient data structure is necessary.

2) Deletion Cache: To manage the addresses and priority of entries that we pseudo-delete, we design an efficient data structure named deletion cache. There are two challenges to be solved for this data structure: (1) quick lookup of the address of deleted entries of specific priority; (2) record the available addresses of entries in minimal space.

TCAMs are expensive and power-hungry; thus, the TCAM size on a switch is quite limited (2000–3000 entries). Because

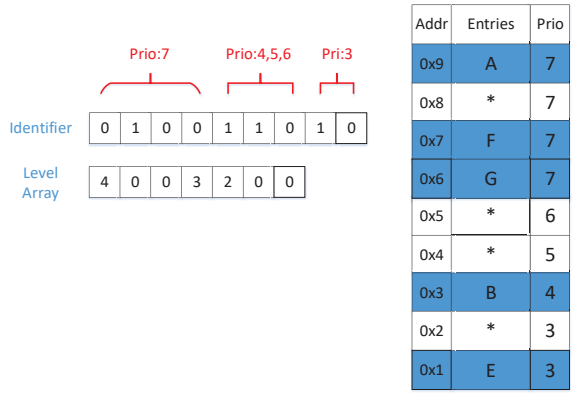


Fig. 11. An example of the deletion cache managing pseudo-deleted entries. The table shows the status of TCAM, where white entries have been pseudo-deleted. The level array records the number of entries of each priority. The identifier records the position of the available deleted entries. If we need to insert a rule of priority 7 in the flow table, we first get the rule number of priority 7 in the level array and then the leftmost 4 bits of the identifier ("0100"). These bits tell us that address 0x8 is available.

of this, the number of priorities generated by flow dependencies is not too big in real-life scenarios (less than 100). Our deletion cache is based on these observations.

The deletion cache contains an identifier of n binary bits and an array of m characters. Here, n is the number of entries in the TCAM and m denotes the max priority in the scenario. Fig. 11 shows the architecture of the deletion cache. According to the level array, we can get a binary string from the identifier representing rules of specific priority. The position of a one in the string tells us the address of an available entry. There have been some efficient algorithms to get the rightmost one of a short binary string based on bit operations in $O(1)$ [19]. Therefore, the time complexity of the optimized addition operation is $O(P)$ (P is the number of priorities in the TCAM).

Algorithm 2 shows the complete maintenance and search algorithm of our deletion cache. During cache operation, the priority of the available entries decreases gradually, which improves performance because of the greater number of moves needed by lower-priority insertions. Our basic RT-S algorithm is based on the deletion cache. When the switch executes a deletion instruction, the target entry is pseudo-deleted and added to the deletion cache. When executing an addition instruction, the switch searches for an available entry in the deletion cache and execute a modification. Otherwise, the addition instruction is executed in its original form.

3) MFD-Based Deletion Cache: As mentioned in Section II, MFD of modification will influence the instruction execution time. Therefore, we further improve our deletion cache by returning the entry of the minimum MFD with the inserted rule. To support minimum MFD match, we need a cache to record elements of every match field in the deleted entries and return the entry having the most same-match-field elements. Inspired by the design of Locality Sensitive Hashing (LSH) [20], we a propose Minimum Distance Deletion Cache (MDDC) to solves the challenges above and support our optimized RT-S algorithm.

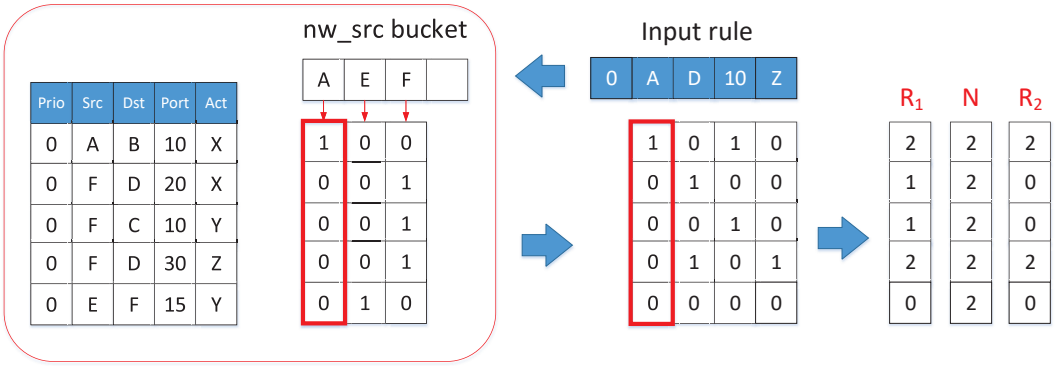


Fig. 12. An example of searching in MDDC for an appropriate entry, which has minimum MFD and the same priority as the incoming instruction. There are five available entries of priority 0 in the TCAM. For each match field, a bucket stores the identifier of every element having appeared. Due to space limitation, we do not show buckets of all fields here. As an example, the new_src bucket stores the identifiers of three elements: A, E, F. When a rule is inserted, we get four identifiers from four buckets and sum their interval of priority one (R_1). For the approximate solution, we AND R_1 and N (assuming that the minimum standard is set as 2) to get R_2 and then return the rightmost one in R_2 (which is Entry 3).

Algorithm 2: Basic RuleTailor-S

Data: level array LA, identifier ID, target rule R, capacity of TCAM N, the position P of R in the TCAM, and the instruction I

function *Cache_Adding(R,P)*:
  $n \leftarrow 1 << (N - P)$
  $ID \leftarrow ID + n$
  if *there is no rules of Priority R.prio* **then**
    move the record in $LA[R.prio]$ backward
  end
end
function *Cache_Popping(R)*:
  $\forall r\ s.t.\ r.prio > R.prio,\ n_b \leftarrow \sum LA[r.prio]$
  $p \leftarrow R.prio$
  while $LA[p]=0$ **do**
    p -= 1
  end
  $n_e \leftarrow LA[p]$
  $B \leftarrow$ the bits from n_b to n_e
  $res \leftarrow$ the rightmost one's position in B
  return $res + n_b$
end
if $I.type = DEL$ **then**
  Cache_Adding($I.rule$, $I.pos$)
end
if $I.type = ADD$ **then**
  $res \leftarrow Cache\_Popping(I.rule)$
  if *res exists* **then**
    modify the rule in res
  else
    insert $I.rule$ into TCAM
  end
end

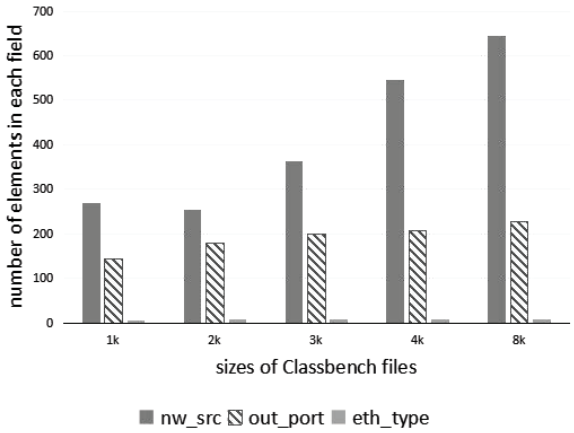


Fig. 13. Element numbers of match fields.

In MDDC, each element of the match field belongs to a specific identifier, which is similar to the one in an original deletion cache. The difference is that we use more than one bit to represent an entry in the TCAM. For example, OpenFlow v1.0 supports 12 match fields, so we use four bits as a cell to count the number of same-match fields. When searching for an available entry, we just need to sum 12 corresponding intervals of identifiers given by the level array and find the cell of the maximum value in the interval. The time complexity of this algorithm is $O(P+S)$ (P is the number of priorities in the TCAM, and S is the number of entries of a specific priority). There is also an approximate solution to find the best-matched entries in $O(P)$: after summing the 12 identifiers we can AND (or compare) the result with a specific binary string N, which is set according to the expected lower bound of matched number and has the same bits as the identifier does. This operation will filter out all entries that match fewer fields than the standard. For example, if we set the minimum standard as 8 matched, N is made up of the cell 1000. Then, we can get the rightmost one of the result as we do in the original deletion cache. In Fig. 12, we show the design of MDDC in detail.

To determine the space complexity of MDDC, we count

the number of elements in several main match fields of rules generated by Classbench. As shown in Fig. 13, when the TCAM capacity ranges from 2000 to 3000, the element number is usually less than 500. To support our MDDC we need $12 \times 500 \times 2000 \times 4 = 4.8 \times 10^7 bits = 6MB$, which is small.

Algorithm 3: RuleTailor-H

Data: instruction I, the target IS of size n, execution time in specific priority order: ADD, MOD, DEL
function *RT-H_CONTROL(IS)*:
  Input: the target IS of size n, execution time in specific priority order: ADD, MOD, DEL
  Result: three IPQs of different operation types
  $Pattern \leftarrow$ {$pattern\ of\ best\ order\ for\ each\ operation$: [$pattern1 : ASC\_DEL + ASC\_MOD + ASC\_ADD$] [$pattern2 : ASC\_DEL + ASC\_MOD + DSC\_ADD$] ...}
  for $i \in IS$ **do**
    $IPQ_{i.type} \leftarrow i$
  end
  return $IPQ_{DEL}, IPQ_{MOD}, IPQ_{ADD}$
end
function *RT-H_DATA(I)*:
  if $I.type = ADD$ **then**
    if *extra space available* **then**
      $P \leftarrow MDDC.pop()$
      modify the rule at P into $I.rule$
    else
      insert $I.rule$ into TCAM, update $MDDC.LA$
    end
  end
  if $I.type = DEL$ **then**
    pseudo_delete $I.rule$
    $MDDC.add(I.rule, I.pos)$
  end
  if $I.type = MOD$ **then**
    modify the target rule into $I.rule$;
  end
end
$IS \leftarrow HIO\_CONTROL(IS)$
for $i \in IS$ **do**
  $HIO\_DATA(i)$
end

C. Hybrid Instruction Optimization

In the RT-S algorithm, we utilize observations from Section II to reduce unnecessary moves in insertion and deletion and execute modification operation with less MFD. However, some other observations used in the RT-M algorithm are not considered in RT-S. We notice that both RT-S and RT-M take advantage of the combination of insertion and deletion.

To further optimize our hardware flow update mechanism we combine two algorithms and design a flow update algorithm working both in the data and control plane. This RT-H algorithm for hybrid optimization, is shown as Algorithm 3 and achieves higher performance for hardware flow updates. In RT-H, the IS is classified by the instruction type and sorted by the priority in the control plane according to measurement results. Then, as we do in RT-M, the IS is sent to switches in the type order of DEL, MOD, ADD. However, no integration or transformation happening during the preprocessing since we have more efficient methods in the data plane to deal with deletion operations. When executing the IS, we use pseudo deletion instead of the traditional deletion and our MDDC still works in RT-H to manage the extra free space. Thus, the preprocessing in the control plane boosts the utilization of extra free space. In our evaluation in the following section, we can see that RT-H performs the other algorithms in all scenarios.

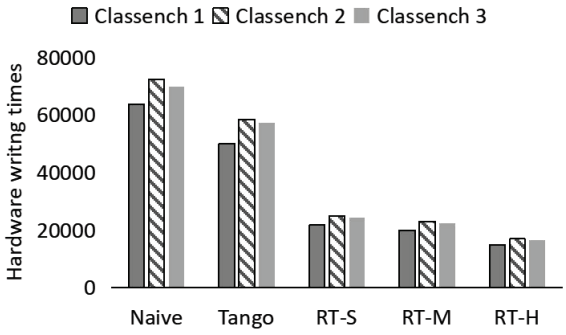


Fig. 14. Hardware write times for various Classbench files.

V. Evaluation

In this section, we present the evaluation results of RuleTailor and demonstrate that using our optimizing approaches, we can achieve higher flow update performance. As mentioned above, we use RT-M, RT-S, and RT-H to indicate the three different types of RuleTailor algorithms.

A. Experimental Environment

We use a software TCAM to emulate flow updates and count the number of TCAM entries moves. As each TCAM move costs a constant amount of time (0.6ms) [9], the total number of TCAM moves times the average latency of a TCAM move can be used to estimate the TCAM update time. We use access control lists from Classbench [21] to generate flow rules with dependencies and flow update instruction sets. To avoid dependency among instruction sets, we choose origin rules in the flow table as the target of modification and deletion instructions.

B. Methodology

To abstract the flow update behaviors of different switches and demonstrate the performance of our algorithms, we execute our flow update emulation for different operation proportions, TCAM sizes, and batch sizes to assess the running time of algorithms. We set the priorities of generated rules by topological dependency (we use topological sorting to obtain

TABLE III
PRIORITY NUMBER, INSTRUCTION TYPE DISTRIBUTION PER CLASSBENCH FILE

Flow files	Priority number	Insertion number	Modification number	Deletion number
1	11	294	97	106
2	15	269	98	90
3	10	292	109	97

rule dependency and then use the same priority number for rules that have no dependencies with each other). The default preferred execution order is ascending and we suppose that inserting and deleting a rule in the flow table takes more time than modification (the same as in most switches). We define N as the capacity of the virtual TCAM, and there are $N/2$ rules in the TCAM originally and $N/2$ instructions to be processed in the experiment. The default field number is 5 and the default batch size is $N/2$.

C. Hardware Write Times

Since matching an entry in the TCAM is much faster than writing, the primary bottleneck of flow updates is the delay of hardware writes. In this experiment, we evaluate different flow update algorithms in a microcosmic perspective by counting the total TCAM entry writing times of addition, deletion and modification. We execute this experiment in a default scenario: the capacity of the TCAM is 1k and the proportion of three types of operations is approximately 3:1:1 (I:M:D). We repeat the above experiment three times using three different Classbench files. The main parameters in this experiment are shown in Table III.

In this experiment, we choose the naive update algorithm (move the rules one-by-one when inserting and deleting) as our baseline and compare the performance of RT-S, RT-M, and RT-H algorithms. Additionally, we compare the performance of our update patterns with Tango[14], a network update scheduler for SDN. The core of Tango uses a scheduling algorithm that constantly scores the instructions and chooses the rule of the highest score to be pushed first. However, unlike RuleTailor, Tango does not take into account the possibility of operation transformation and integration, which expand the optimization space for flow updates.

As shown in Fig. 14, we can observe that compared with naïve algorithm, our algorithms have much better performance in hardware times: RT-S 66%, RT-M 68% and RT-H 77%. In addition, our algorithms also outperform Tango: RT-S 57%, RT-M 61% and RT-H 71%. As RuleTailor transforms additional operations into modifications, it requires more modification operations to finish an update. But considering that the number of extra modification is much less and modification operations are faster than insertion (according to our hypothesis), this cost is small. The pseudo-deletion used in RT-S and RT-H reduces the hardware write times for deletion compared to Tango and the naïve algorithm. And for RT-M, nearly all deletion operations are combined with insertions. The only exception is the case of Classbench 2 where the priority structure of the instruction set is more complicated and thus, for some deletion instructions, there is no additional instruction of the same priority that can be combined.

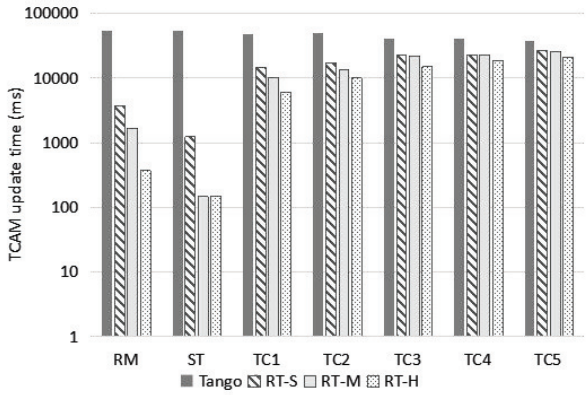


Fig. 15. Flow update time in different scenarios.

D. Operation Proportion

The operation proportion of the IS has a big influence on our RuleTailor algorithms. Here, we define the proportion of addition operations and deletion operations as the I-D rate. For some specific scenarios, the I-D rate is different from the baseline and fits RuleTailor better. To further evaluate the performance of our algorithms, we vary the I-D rate of our instruction sets to investigate the efficiency changes of the algorithm in different scenarios. To determine the I-D rate in each experiment, we consider several real-life scenarios: (1) RM (rule migration): to reroute flows because of load balance or device failure, the number of former control policies is nearly the same as the latter. (2) ST (network state transition): to control the state of the network to which the switch belongs [22], we delete several flows and insert similar ones consistently. The number of additions is the same as deletions and both instructions are executed alternately. (3) TC (traffic change): to change one traffic path in the network a number of flows are updated in the flow table. We have implemented several TC scenarios (denoted as TC x) with different proportion on the number of instructions of each type. For convenience, the addition operations in TC x are $x+1$ times as many as deletions.

Fig 15 shows the running time of different algorithms in RM, ST, and TC x scenarios. In these specific scenarios of lower I-D rate, our algorithms perform much better. Compared to Tango, RT-H get an improvement of 99% in RM, 99% in ST, and 44%-87% in TC x; RT-M get 97% in RM, 99% in ST, 32%-79% in TC x; RT-S get 93% in RM, 98% in ST, 30%-70% in TC x. In ST, since the rules inserted are exactly similar to the deleted ones, all addition operations can be transformed into modifications and the only hardware writes are from modifications and deletions. Therefore, the performance is further improved. From Fig. 16, we can observe that our algorithms perform increasingly better as the I-D rate decreases and achieve the highest performance in ST.

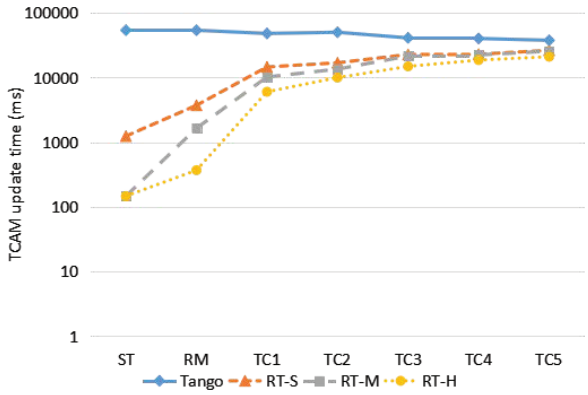


Fig. 16. Flow update time over I-D rate.

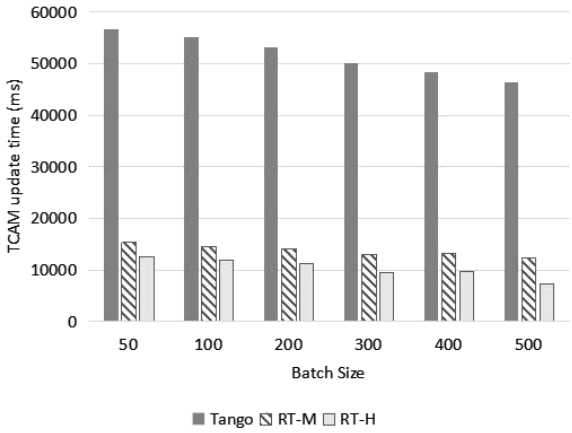


Fig. 18. Flow update time for different batch sizes.

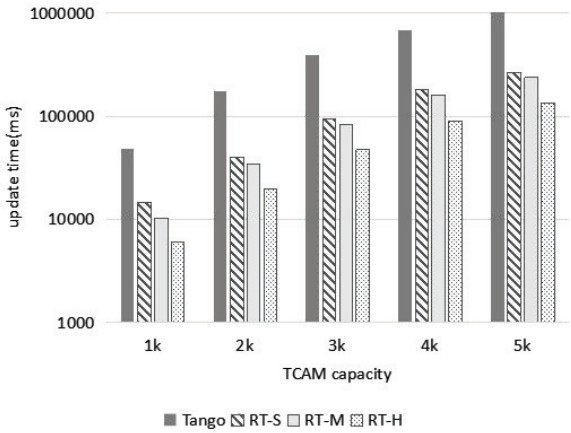


Fig. 17. Flow update time for different TCAM capacities.

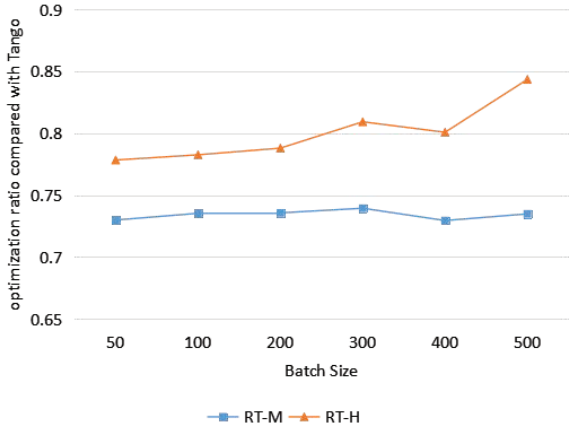


Fig. 19. Optimization ratio over batch sizes compared with Tango.

E. Space Scalability

To confirm that our methods are robust and scalable, we repeat the flow update experiment four times using four Classbench files of different size and measure the running time of different execution patterns. The I-D rate in this experiment is set to 2:1 in a more normal situation (TC) to observe the performance.

Compared with Tango, we observe 87%–88% improvement varying with instruction number (500–2500) for RT-H, which is a robust result. In contrast, the performance of the other two algorithms is less stable: RT-M varies from 76% to 80% and RT-S varies from 70% to 76%, both of which are worse than RT-H overall.

F. Batch Size

In the RuleTailor framework, we use a batch to collect *IS* from incoming instructions in the control plane (as Tango does). Thus, for optimization algorithms working in the control plane, such as RT-M and RT-H, the performance varies with the batch size. In this experiment, we divide the *IS* into several subsets and use different algorithms to preprocess them separately. Then, we compare the total update time of RT-M, RT-H and Tango. We conduct this experiment in a default scenario where a TCAM capacity of 1k and a proportion of operations approximately 2:1 (I:D). Fig. 18 shows the optimization ratio of update time compared with Tango.

As shown in Fig. 18, all three algorithms perform better as the batch size grows, but the rank of algorithm efficiency does not change, which is RT-H < RT-M < Tango in total update time. Fig. 19 shows that RT-H's improvement increases even faster than the other two algorithms, from 78% to 84% compared with Tango. This means that with the development of flow update technology, RuleTailor can achieve higher efficiency since the number of instructions that can be pre-processed in a batch will increase.

VI. Related Work

In this section, we review existing state-of-the-art technology for OpenFlow switch flow update. We can see that RuleTailor makes unique and novel improvements inspired by previous schemes.

A. Switch Diversity Exploration

Previous studies have aimed to understand the diversity of switch implementations. By making detailed measurements on

commercial off-the-shelf switches [5, 12, 23], we have found significant differences in flow table capacity, flow update time, and other attributes. To address this diversity, some models have proposed to abstract the existing switch architectures [13, 24]. Some researches have tried to adapt diverse switches and other network elements to a uniform and abstract scenario through standard protocols (like OF-config), programming languages [25], or compilers [18]. On the other hand, Tango [14] and Mazu [26] try to accept the diversity and make use of it in the optimization framework. RuleTailor carries on this idea and promotes a new mechanism to quantify and utilize switch diversity.

B. Flow Update in Control Plane

There have been several approaches that can reduce the cost of flow update in the control plane. For example, Tango [14] try to get a better update instruction sequence to execute, which shortens the delay of flow update. And some schemes [6, 8, 17] cut down the number of instructions sent to switches by rule combination. In a wider range of total network, Dionysus [27] and other technologies [7] reduce the multi-switch delay by combining and filtering instructions in a global view. While RuleTailor takes into consideration the diverse instruction execution time of commercial off-the-shelf switches and thus can be more adaptive in practical applications.

C. Flow Update in Data Plane

Typically, flow update efficiency in the data plane can be improved by designing new firmware with better algorithms. Some architectures, such as TCAM update cache [28] and hierarchical flow table [29], have been proposed and come into use. In recent years, partial order theory is proposed [9, 10, 30], which gives the lower bound of TCAM update. Compared with them, RuleTailor is simpler and easier to implement and can be combined with other state-of-the-art technologies.

VII. Conclusion and Future Work

Our measurement results show the implementation diversity of OpenFlow switches when considering flow updates. This difference leads to the varying performance of update instruction execution. To speed up flow updates considering this diversity, we present RuleTailor, an optimization framework for hardware flow updates. RuleTailor investigates the flow update behavior of target switches using our measurement framework. Then, RuleTailor chooses different algorithms to speed up flow updates through instruction transformation, integration, and priority sorting, according to measurement results. RuleTailor also optimizes the TCAM update behavior in the data plane through pseudo-deletion and a special data structure named minimum distance deletion cache. We analyze the efficiency of RuleTailor under different conditions. The results demonstrate that RuleTailor can significantly reduce the delay in hardware flow updates.

For our future work, RuleTailor shows compatibility with other state-of-the-art flow update technologies, such as the DAG-based memory update algorithm [9, 10], the partial order theory [30], and free space obligation [11, 31]. Finally, RuleTailor optimizes single switch flow updates independently, but if we take the multi-switch network update optimization [27, 32, 33] into consideration, more improvement might be possible to further reduce the delay of updates.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[2] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks." in *Proc. USENIX NSDI*, vol. 10, no. 8, 2010, pp. 89–92.

[4] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an mpls transport profile," Tech. Rep., 2009.

[5] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini, "Methodology, measurement and analysis of flow table update characteristics in hardware Openflow switches," *Comput. Netw*, vol. 136, pp. 22–36, 2018.

[6] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proc. USENIX NSDI*, 2015, pp. 87–101.

[7] S. Vissicchio, L. Cittadini, S. Vissicchio, and L. Cittadini, "Safe, efficient, and robust SDN updates by combining rule replacements and additions," *IEEE/ACM Trans. Netw*, vol. 25, no. 5, pp. 3102–3115, 2017.

[8] K. Kannan and S. Banerjee, "Compact TCAM: Flow entry compaction in TCAM for power aware SDN," in *Proc. ICDCN*. Springer, 2013, pp. 439–444.

[9] X. Wen, B. Yang, Y. Chen, L. E. Li, K. Bu, P. Zheng, Y. Yang, and C. Hu, "RuleTris: Minimizing rule update latency for TCAM-based SDN switches," in *Proc. ICDCS*. IEEE, 2016, pp. 179–188.

[10] K. Qiu, J. Yuan, J. Zhao, X. Wang, S. Secci, and X. Fu, "FastRule: Efficient flow entry updates for TCAM-based OpenFlow switches," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 3, pp. 484–498, 2019.

[11] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, 2001.

[12] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. PAM 2015*. Springer, 2015, pp. 347–359.

[13] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *Proc. HotSDN*. ACM, 2013, pp. 43–48.

[14] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu, "Tango: Simplifying SDN control with automatic switch property inference, abstraction,

and optimization," in *Proc. CoNEXT*. ACM, 2014, pp. 199–212.
[15] H. Chen and T. Benson, "Hermes: Providing tight control over high-performance SDN switches," in *Proc. CoNEXT*. ACM, 2017, pp. 283–295.
[16] H. Song and J. Turner, "Nxg05-2: Fast filter updates for packet classification using TCAM," in *IEEE GLOBECOM*. IEEE, 2006, pp. 1–5.
[17] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu, "Compiling minimum incremental update for modular SDN languages," in *Proc. HotSDN*. ACM, 2014, pp. 193–198.
[18] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak, "Maple: Simplifying SDN programming using algorithmic policies," in *Comput. Commun. Rev.*, vol. 43, no. 4. ACM, 2013, pp. 87–98.
[19] Y. Li, J. M. Patel, and A. Terrell, "Wham: a high-throughput sequence alignment method," *ACM Trans. Database Syst.*, vol. 37, no. 4, p. 28, 2012.
[20] L. Paulevé, H. Jégou, and L. Amsaleg, "Locality sensitive hashing: A comparison of hash function types and querying mechanisms," *Pattern Recognition Letters*, vol. 31, no. 11, pp. 1348–1358, 2010.
[21] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, June 2007, pp. 499–511.
[22] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful Openflow applications inside the switch," *Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.
[23] P. Rygielski, M. Seliuchenko, S. Kounev, and M. Klymash, "Performance analysis of SDN switches with hardware and software flow tables," in *Proc. EAI International Conference on Performance Evaluation Methodologies and Tools*, 2016.
[24] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for OpenFlow switch evaluation," in *Proc. PAM 2012*. Springer, 2012, pp. 85–95.
[25] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.
[26] K. He, J. Khalid, S. Das, A. Akella, E. L. Li, and M. Thottan, "Mazu: Taming latency in software defined networks," Tech. Rep., 2014.
[27] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Comput. Commun. Rev.*, vol. 44, no. 4. ACM, 2014, pp. 539–550.
[28] H. Chen and T. Benson, "The case for making tight control plane latency guarantees in sdn switches," in *Proc. ACM SOSR*. ACM, 2017, pp. 150–156.
[29] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proc. ACM SOSR*. ACM, 2016, p. 6.
[30] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast TCAM updates," *IEEE/ACM Trans. Netw*, vol. 26, no. 1, pp. 217–230, 2018.
[31] B. Vamanan and T. Vijaykumar, "TreeCAM: decoupling updates and lookups in packet classification," in *Proc. CoNEXT*. ACM, 2011, p. 27.
[32] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proc. ACM SIGCOMM*, vol. 43, no. 4, 2013, pp. 15–26.
[33] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," in *Proc. ACM SIGCOMM*, vol. 43, no. 4, 2013, pp. 411–422.

Bohan Zhao is a bachelor student in Fudan University. His research interests include computer networking and computer architecture.

Jin Zhao received his B. Eng. Degree in computer communications from Nanjing University of Posts and Telecommunications, China, in 2001, and the Ph.D. Degree in computer science from Nanjing University, China, in 2006. He joined Fudan University in 2006. He stayed at University of Massachusetts Amherst for 1 year as a visiting scholar in 2014. His research interests include software defined networking and network function virtualization. He is a member of IEEE and ACM.

Xin Wang received his Bachelor Degree and the Master Degree from Xidian University, Xi'an, China, in 1994 and 1997 respectively, in Information Theory and Communications. He received the Ph.D. Degree from Shizuoka University, Japan in 2002, in Computer Science. Since 2002, he has been with the School of Computer Science at Fudan University, where he is currently a full professor. He is a member of IEEE and CCF.

Tilman Wolf is Professor of Electrical and Computer Engineering and Senior Vice Provost for Academic Affairs at the University of Massachusetts Amherst. He received a Diplom in informatics from the University of Stuttgart, Germany, in 1998. He also received a M.S. in computer science in 1998, a M.S. in computer engineering in 2000, and a D.Sc. in computer science in 2002, all from Washington University in St. Louis.

Dr. Wolf is engaged in research and teaching in the areas of computer networks, cybersecurity, and embedded systems. He is a senior member of the IEEE and member of the ACM. He has been active as program committee member and organizing committee member of numerous professional conferences, including IEEE INFOCOM and ACM SIGCOMM.