CURE : Consistent Update with Redundancy Reduction in SDN

Ilora Maity, Student Member, IEEE, Ayan Mondal, Student Member, IEEE, Sudip Misra, Senior Member, IEEE, and Chittaranjan Mandal, Senior Member, IEEE

Abstract—In this paper, we address the issue of rule duplication during network information updates in Software Defined Networking (SDN). In OpenFlow-enabled SDN, network update involves the controller in sending update packets to desired set of switches, where the update rules are installed. To ensure update consistency, old flow rules are stored until the total update procedure is complete. In worst case scenario, each switch requires storage space of two times than the size required for original number of rules. Furthermore, majority of the OpenFlow switches have expensive ternary content addressable memories (TCAMs). Higher consumption of TCAMs during update increases cost of network update and decreases scalability of SDN. Moreover, non-availability of storage space within update duration triggers packet drop. In this work, we propose an approach for consistent update with redundancy reduction, named CURE, that reduces TCAM usage during update. CURE prioritizes switches according to their usage pattern and schedules update based on priority zones. The proposed approach guarantees that highly loaded switches are updated first. CURE also maintains packet-level consistency by implementing a multilevel queueing approach. In this framework, each switch, in current update region, stores incoming packets in individual device queues until it completes update. Therefore, after initiation of update, packets are processed according to new rules only. We implemented our scheme in Matlab environment. Average rule space utilization during update in CURE is 29.954% less than the two-phase update proposed in existing literature.

Index Terms—SDN, Network Update, Big Data, OpenFlow, TCAM, Multiclass Classification, Queueing Theory.

I. INTRODUCTION

In traditional networks, each network device or switch includes both a control plane and a data plane. Control plane determines the forwarding rules for incoming packets. Data plane stores the forwarding table, which is a collection of the forwarding rules determined by control plane. Therefore, traditional networks are complex and resistant to changes. Software Defined Networking (SDN) is a new networking paradigm which separates control plane and data plane to provide network services dynamically [1]. In SDN, a single controller or a cluster of controllers determines forwarding rules and installs them to the switches. Switches are only forwarding devices which store the forwarding tables and forward incoming packets based on matched table entries. With the rapid increase in data volume, dynamic nature of SDN gets attention of researchers for implementing big data solutions [2], [3]. Features of SDN such as — detachment of control plane from data plane, programmable network, and availability of a global view of the whole network to a centralized control plane, make SDN an attractive choice for designing the backbone of collecting, processing, storing, and analyzing big data.

Similar to other networks, updates in SDN occur frequently. Major reasons for network update are — (1) optimization of flow table, (2) flow swapping after arrival of new flows, (3) traffic monitoring, (4) maintaining state of shut down switches, (5) expiration of forwarding rules’ time-out, and (6) switch or link failure [4]. Network update in traditional network involves changing configuration of each switch, separately. On the other hand, SDN update is triggered by the controller which generates forwarding rules for new network configuration and installs those rules to the required switches. Additionally, controller performs garbage collection by deleting old rules [5].

Presently, most of the OpenFlow switches available in the market have Ternary Content Addressable Memory (TCAM). TCAMs are high-speed memories which can match flow rules in parallel in $O(1)$ time. Each TCAM entry is a ternary string consisting of 0, 1, and * (don’t care) having components are match fields, priority, counters, instructions, timeouts, cookie, and flags [6], [7]. Based on a forwarding rule, a single or multiple TCAM entries can be generated [8]. In a simple rule, there is an one-to-one mapping between each value and match fields in a flow table entry. Whereas, for range rules, some values such as port numbers are denoted by ranges. Therefore, when transformed to ternary values, these rules generate multiple TCAM entries [9]. SDN update is basically addition of new TCAM entries and removal of older entries from one or more switches [10]. Therefore, update of a single forwarding rule causes update of a single or multiple TCAM entries.

Existing network update techniques for SDN are centered on four basic approaches such as Ordered Update, Incremental Update, Timed Update, and Buffered Update. Ordered update [11] is a sequence of update phases defined by the controller. Each phase is executed after completion of previous phase. In incremental update [12], network is updated in multiple phases, where each phase updates a subset of switches. Each switch maintains a record of both old and new configurations until the total network gets updated. After completion of update, flow table entries for old configuration are deleted. Packet level consistency is maintained by ensuring that each packet follows either the new configuration or the old configuration. Both of these approaches require extra flow table
space for accommodating duplicate rules. Moreover, controller is involved until the completion of update for all the switches. To reduce this overhead, Mizrahi et al. [5] proposed a timed consistent network update scheme where updates are scheduled at specific time orders. This approach reduces duration of storing duplicates rules in switches. Controller is only involved at the beginning of update when it forwards update instructions to all the switches. The switches execute updates according to the predefined schedule. In buffered update [13], incoming packets are buffered at the control plane until the total update procedure is completed. This approach ensures both packet-level consistency and flow-level consistency. However, controller load increases in this method. Therefore, none of the existing SDN update approaches consider complete elimination of redundant rules. In this work, we propose a SDN update policy without storing redundant rules.

A. Motivation

Presently, OpenFlow is the most favoured protocol for implementing SDN [14] [15]. Expensive TCAMs in OpenFlow-based SDN switches consume a large amount of power [16], [17] and occupy large footprints [18]. Power consumption in TCAM is 100 times more than RAM storage per Mbit and same amount of TCAM costs 400 times more than RAM storage [19]. These constraints restrict storage capacity of TCAMs [20]. However, existing SDN update policies require storage of old configuration rules until the whole update process is completed. These approaches require maintaining extra storage space which leads to high cost. Hence, for the worst case scenario 50% of the storage space needs to be empty before starting network update. Therefore, cost of storing redundant rules decreases scalability of overall network. Furthermore, number of big data-based applications are increasing. Continuous flow of high-volume data generates high number of update entries for each flow table. Therefore, providing storage space for both old and new TCAM entries can be a bottleneck for big data applications.

In this paper, updates in SDN switches are scheduled in an optimized manner so that high priority switches are updated first. We build a priority index for all the switches based on the frequency of matched rules. The packet level consistency is also ensured by employing a packet-queueing mechanism with minimum service latency.

B. Contribution

Our work aims to minimize TCAM usage for efficient processing of big data. The primary contributions of our work are listed below.

1) Initially, we determine the priorities of all the SDN switches in the network. Each flow table entry in an Openflow switch maintains a record of total number of packets matched with that particular entry in past. We used this record as input of a multiclass classification algorithm to determine priorities for each switch.

2) We design a priority-based algorithm for scheduling updates to SDN switches.

3) Finally, we propose a packet queueing mechanism for maintaining consistency of incoming packets during update.

C. Paper Organization

The remainder of this paper proceeds as follows. Section II discusses the existing approaches for SDN update. In Section III, we define the network model and describe the proposed scheme. Section IV depicts the experimental results and comparative studies with other existing approaches. Finally, Section V concludes the work.

II. RELATED WORK

This section gives an overview of the existing literature related to network update in SDN. Existing literature in this field can be categorized in four parts such as — Ordered update, Incremental update, Timed update, and Buffered Update.

A. Ordered Update

In case of ordered update, controller partitions the total update procedure into multiple stages. It waits for completion of each stage before starting the next stage. The last stage is garbage collection stage where older rules are deleted.

Updating network in an ordered manner is a well-researched topic in networking domain. Francois et al. [11] proposed an ordered update sequence for forwarding devices. The proposed scheme ensures consistency by preventing loop. It also allows rerouting of packets during failure of links or forwarding devices. However, this approach requires modification of network protocols as well as forwarding devices. Clad et al. [21] generated an optimized sequence for updating weights of links. The proposed greedy algorithm is free from loops and does not require modification of network protocols. Similar approach is proposed in Ref. [22] which considers configuration update of Interior Gateway Protocols (IGPs). The proposed approach encounters a large overhead due to storage of two IGP configurations simultaneously. Thereby, ordered update policy in SDN encounters service latency as update of each phase is restricted by completion of update of previous phase.

B. Incremental Update

In incremental update approach, network is updated in multiple phases or rounds where each round updates a portion of flow rules or a subset of switches.

Reitblatt et al. [12] proposed a two-phase update approach where all the internal switches are updated in the first phase. Next, the ingress switches are updated. Each updated ingress switch attaches a new version tag to each of the incoming packets. The switches maintain both the old and new rules. The incoming packets are processed by either of them (not both) only based on the version tag. This approach ensures packet-level consistency. Older rules are deleted only when there is no packet with old version tag existing in the network. This method increases load on ingress switches as they have to change the incoming packets. Moreover, memory overhead
is incurred for storing old rules. In another work, Canini et al. [23] also discussed similar approach. This work implements an update approach similar to database transactions where either all switches are updated or none. If any switch fails to update during the procedure, all the previous updates are rolled back.

C. Timed Update

Mizrahi et al. [5] proposed an extension of OpenFlow protocol by scheduling the update phases at particular time instants for both ordered and incremental updates. This approach preserves packet level consistency by avoiding conflicts in updates. Controller’s involvement during update procedure is also reduced, as the controller sends update packets to all the switches at the beginning of the update with attached timestamp value of scheduled update. This technique reduces the duration of update as well as the duration required to store older rules in SDN switches. However, synchronizing updates to all the switches encounters computational complexity and depends on characteristics of particular forwarding devices.

D. Buffered Update

Buffered update approach [13] redirects all the incoming packets arriving at the switches to be updated to the controller. These packets are buffered in the control plane until all the switches are updated. After completion of update procedure, the controller releases these packets, and thereafter the packets are processed according to new configuration. The major limitation of this approach is that it overloads the controller and increases service latency for the affected packets.

All of the existing update approaches require storage of older rules until the update procedure completes. As OpenFlow switches have expensive TCAMs, storage of duplicate rules turns out to be a major limitation, specially, when processing of big data is required.

III. THE PROPOSED SCHEME

In this section, we describe the network model considered for our proposed scheme, CURE. We indicate the different symbols in Table I. We also discuss the approach for implementing redundancy-free consistent update of SDN.

A. Network Model

We model the network as a graph \( G = (N, L) \), where \( N \) is the set of nodes, and \( L \) is the set of links between the nodes. The set \( N \) can be expressed mathematically as:

\[
N = C \cup S,
\]

where \( C \) is the set of controllers, and \( S \) is the set of OpenFlow switches. Figure 1 shows the proposed network model. The upper bound of the number of flow rules which can be stored in an OpenFlow switch \( S_i \) is denoted as \( U_i \). A subset \( S^{\text{ingress}} \) of set \( S \) is considered as ingress switches which have one or more ingress ports for receiving incoming packets. Each switch \( S_j \) has an associated device queue denoted as \( Q_j \). The set of immediate neighbors of a switch \( S_j \) is denoted by \( \text{neighbor}(S_j) \). On the other hand, the set of links \( L \) can be defined as:

\[
L = L_{cc} \cup L_{cs} \cup L_{ss},
\]

where \( L_{cc} \) is the set of link between the controllers, \( L_{cs} \) is the set of control links between the controllers and the OpenFlow switches, and \( L_{ss} \) is the set of data links between the OpenFlow switches for packet forwarding.

For simplicity, we assume a centralized control plane consisting of only a single controller \( C \). Hence, \( S \) can be defined as \( S = \{S_1, S_2, ..., S_{|N|-1}\} \). \( L_{cs} = \phi \) and number of links in \( L_{ss} \) is \( |S| = |N| - 1 \). Each link \( L \) has an associate link capacity \( c_l \).

![Fig. 1: SDN architecture](image)

Centralized controller installs forwarding rules in the switches. These forwarding rules define the path of a flow from source to destination without exceeding capacity of each link. The set of incoming flows is denoted by \( F \). We define each flow \( F_a \) as a tuple \( <a, Path_a, S(F_a)> \), where \( a \) is the flow id, \( Path_a \in S \), which is the sequence of nodes, defines the flow path, and \( S(F_a) \in S^{\text{ingress}} \) is the ingress switch for flow id \( a \).

Each switch stores forwarding rules in one or multiple flow tables [7]. Each flow table entry is a ternary string. Therefore, a flow rule \( R_i \) in \( S_j \) can be represented as:

\[
R_i \in FT^j_0 \cup FT^j_1 \cup ... \cup FT^j_{n-1},
\]

where \( FT^j_m \) is the \( m^{th} \) flow table of an OpenFlow switch \( S_j \) and \( n \) is the total number of flow tables for that switch. The sets of rules in each of the flow tables of \( S_j \) are mutually exclusive. A flow rule \( R_i \) is denoted by a tuple \( <Pr_i, M_i, A_i> \), where \( Pr_i \) denotes rule priority, \( M_i \) denotes set of match fields, and \( A_i \) denotes set of action values. Each flow rule also contains a set of counters for storing the rule statistics, timeout value, cookie, and flags [7]. If an incoming packet matches with multiple rules, then the rule with highest priority value is selected and the corresponding action is taken.

**Definition 1.** The state of a switch \( S_j \) at time \( t \) is defined by:

\[
A_j(t) = \{R_i(t), L^D_{cs}(t), L^D_{ss}(t), \tau^j(t)\},
\]
where $R_i(t)$ is the set of flow rules of switch $S_j$ at time $t$, $L_{cs}(t) \in L_{cs}$ is the set of control links involving $S_j$ at time $t$, $L_{ds}(t) \in L_{ds}$ is the set of data links involving $S_j$ at time $t$, and $\tau_j$ is the last update time of $S_j$ at time $t$.

**Definition 2.** The network configuration at time $t$ is defined by:

$$\Gamma(t) = \{\mathcal{N}(t), \bigcup_{j=1}^{[S]} \Lambda_j(t)\},$$

(5)

where $\mathcal{N}(t) \in \mathcal{N}$ is the set of nodes at time $t$.

**Definition 3.** Network update in SDN is migration from one network configuration $\Gamma$ to another configuration $\Gamma'$ such that:

$$\Gamma(t_i) \neq \Gamma'(t_j),$$

(6)

where $t_i \neq t_j$.

Major objective for this work is to minimize TCAM usage during update without congesting the links and to maintain packet level consistency. This optimization problem can be formulated as follows:

Minimize $\sum_{j=1}^{[S]} size(R_i^j)$,

(7)

where $size(R_i^j)$ denotes the amount of TCAM used by rules in $S_j$. Equation (7) computes the total TCAM used by all the switches in the network to store flow rules. Subject to the following constraints:

$$size(R_i^j) \leq size(U_j), \forall S_j \in \mathcal{S}$$

(8)

Equation (8) expresses switch capacity constraint for storing flow rules.

$$\sum_{f=1}^{[F]} l_{u,v}^j \leq c_k, \forall u,v \in \mathcal{N}, \forall L_k \in L, link(u,v) = L_k,$$

(9)

where $l_{u,v}^j$ denotes the load of flow $F_j$ on the link connecting the nodes $u$ and $v$ which is denoted by $link(u,v)$. Equation (9) prevents the collective load of all the flows in a link from exceeding the link capacity.

$M_i^j = M_i^j$ and $A_i^j = A_i^j$,

$\forall \Lambda_j(t_i) = \{R_i^j(t_i), L_{cs}(t_i), L_{ds}(t_i), \tau_j(t_i)\}$,

$\forall \Lambda_j(t_k) = \{R_i^j(t_k), L_{cs}(t_k), L_{ds}(t_k), \tau_j(t_k)\}$

$\tau_j < t_i$,

$R_i^j \in R_i^j(t_i)$,

$R_i^j \in R_i^j(t_k)$,

$Duration(R_i^j) < Duration(R_i'^j)$,

(10)

where $Duration(R_i^j)$ is a counter [7], which denotes the elapsed time after installation of the flow rule $R_i^j$. Equation (10) prohibits storage of older and newer versions of a rule in a switch simultaneously.

**B. Redundancy-free Consistent Update**

In this section, we describe the proposed scheme, CURE, for SDN update. Based on workload, we first classify the to-be-updated switches into three priority regions, namely high, medium, and low. Thereafter, we design an algorithm for scheduling updates among the switches of different priority regions. Next, we propose a packet queueing mechanism to maintain packet level consistency during update. Finally, we propose an algorithm for processing the queued packets.

1) **Switch Classification:** Each OpenFlow switch record maintains a counter field which records details of the matching packets. Based on the counter value, we build a training data set. Therefore, we employ One-Vs-All (OvA) multiclass classification algorithm [24] [25] to classify the to-be-updated switches into three priority zones — low, medium, and high. This classification depends on the network topology, packet arrival rate, and existing flows in the network.

2) **Rule Update:** Algorithm 1 schedules update based on the priority zones. Before starting the update, controller sends UPDATE signal at time $T_0$ to mark the set of switches which are to be updated. Controller waits for a $\delta$ time interval before sending the first update packet. Heavily loaded switches are updated first at time $T_{\text{high}} > T_0$. Next, medium priority switches are updated at time $T_{\text{medium}} > T_{\text{high}}$. Finally, low priority switches are updated at time $T_{\text{low}} > T_{\text{medium}}$. During update procedure at a switch, the set of new rules are installed first and older rules are deleted thereafter. In other words, garbage collection at each switch is performed right after completion of update at the corresponding switch.
Definition 4. After $T_0$, a packet is marked old, if it is processed by a switch which is yet to be updated.

Definition 5. After $T_0$, a packet is marked new, if it is processed by a updated switch.

When controller selects a priority region for update, all the old packets in that region are processed before starting the installation of new rules. This ensures that a packet already processed by an old rule, is processed by old rules only. If an old packet reaches an updated switch, the packet is sent to controller for further decision. Similarly, if a new packet reaches a to-be-updated switch which is not in the current update region, the packet is sent to controller for further decision.

Definition 6. Update duration is the time interval between the dispatch of the first update message by the controller, and update completion of the last switch including garbage collection.

Definition 7. An old packet is termed inconsistent if it reaches an updated switch. An new packet is termed inconsistent if it reaches a switch which is not updated and not in current update region.

Algorithm 1 Rule Update Algorithm

INPUT:
1: $S^{low}$ ▶ Set of low priority switches
2: $S^{medium}$ ▶ Set of medium priority switches
3: $S^{high}$ ▶ Set of high priority switches

PROCEDURE:
1: $S' \leftarrow \emptyset$
2: for all $S_j \in S^{low} \cup S^{medium} \cup S^{high}$ do
3: $\text{SIGNAL}(S_j, \text{UPDATE})$ ▶ Controller sends update signal
4: $\text{WAIT}(\delta \text{ ms})$ ▶ Controller waits for $\delta$ milliseconds
5: end for
6: for all $S_j \in S^{high}$ do
7: Process $P_{\text{old}}^{i}$ ▶ Process old packets
8: Insert $R_j$ ▶ Add set of new rules
9: Remove $R_j$ ▶ Remove set of old rules
10: $S' \leftarrow S' \cup \{S_j\}$
11: end for
12: for all $S_j \in S^{medium}$ do
13: Process $P_{\text{old}}^{i}$ ▶ Process old packets
14: Insert $R_j$ ▶ Add set of new rules
15: Remove $R_j$ ▶ Remove set of old rules
16: $S' \leftarrow S' \cup \{S_j\}$
17: end for
18: for all $S_j \in S^{low}$ do
19: Process $P_{\text{old}}^{i}$ ▶ Process old packets
20: Insert $R_j$ ▶ Add set of new rules
21: Remove $R_j$ ▶ Remove set of old rules
22: $S' \leftarrow S' \cup \{S_j\}$
23: end for
24: return $S'$

3) Packet Queueing: Algorithm 2 depicts a queueing mechanism for consistent processing of incoming packets during an ongoing update procedure. The packet queueing algorithm (PQA) is triggered for each to-be-updated switch in the present update region after controller starts update in that region. If the switch has received an UPDATE signal recently, PQA checks whether the switch is already updated. PQA stores the packet if the update process is incomplete in the corresponding switch. Otherwise, the packet is processed.

Packets are stored in the queue of the corresponding switch until the queue gets full. Thereafter, the packets are redirected to the least priority switch belonging to a lower priority region and having free buffer space within one-hop neighbors of the corresponding switch. In this scenario, a switch-identifier flag is added to the packet header specifying the switch id where the packet arrived initially. The packets are buffered at controller when no such neighbor exists. For each switch, we maintain a counter that counts the number of packets stored outside of the switch’s own buffer.

Algorithm 2 Packet Queueing Algorithm

INPUT:
1: $P_{\text{count}}$ ▶ Number of packets buffered outside of $Q_j$
2: $S^*$ ▶ Set of incoming packets at $S_j$
3: $P^*$ ▶ A switch in current update region

PROCEDURE:
1: $\text{visitedNeighbors} \leftarrow \emptyset$
2: for all $P_i^* \in P^*$ do
3: if $S_j \in S^*$ then
4: Process $P_i^*$ ▶ Standard packet processing
5: else
6: $\text{storePacket}(P_i^*, S_j, j)$ ▶ Insert packet to $Q_j$
7: end if
8: end for
9: return $P_{\text{count}}$
10: function $\text{storePacket}(P_i^*, S_k, j)$
11: if $Q_j$ is not full then
12: Store $P_i^*$ in $Q_j$
13: else
14: $P_{\text{count}} \leftarrow P_{\text{count}} + 1$
15: end if
16: if $\text{getNeighboor}(S_j, Reg) \neq \text{Null}$ then
17: Get one-hop neighbor
18: $\text{storePacket}(P_i^*, \text{getNeighboor}(S_j, Reg), j)$
19: else
20: Add flag ($j$) to $P_i^*$
21: Buffer $P_i^*$ at $C$
22: $P_{\text{count}} \leftarrow P_{\text{count}} + 1$
23: end if
24: end if
25: end function
26: function $\text{getNeighboor}(S_j, Reg)$
27: $S_{\text{neighbor}} \leftarrow$ Find least priority unvisited neighbor belonging to a priority region lower than $Reg$
28: $\text{visitedNeighbors} \leftarrow \text{visitedNeighbors} \cup \{S_{\text{neighbor}}\}$
29: return $S_{\text{neighbor}}$
30: end function

4) Packet Processing: After completion of update, each switch triggers the controller informing that it is ready for processing packets. Algorithm 3 describes the procedure of...
processing the waiting-packets. If the triggering switch’s buffer is full, the packet processing algorithm processes the first $K$ packets waiting at the queue of triggering switch itself. Then a portion of the switch’s buffer is reserved for storing the waiting packets with matching switch-identifier flag among its one-hop neighbors. We name this buffer space as secondary buffer. Size of secondary buffer is determined from the available counter value. Packets waiting in the queue of a neighbor switch and/or controller are shifted to secondary buffer. After processing these packets the secondary buffer space is merged with the switch’s original buffer before processing the new ones.

Algorithm 3 Packet Processing Algorithm

INPUT:  
1: $visitedNeighbors$  \(\triangleright\) Set of neighbors having $S_{u}$’s packets  
2: $S_{u}$  \(\triangleright\) Switch which triggered waiting-packet processing

OUTPUT:  
1: $P''$  \(\triangleright\) Set of packets in secondary buffer

PROCEDURE:  
1: if $|Q_{u}| = K$ then  
2: $P' \leftarrow \emptyset$  
3: Process first $K$ packets in $Q_{u}$  
4: for all $S_{j} \in visitedNeighbors$ do  
5: for all $P_{i}^{j}$ in $Q_{u}$ do  
6: if $P_{i}^{j}$ contains $flag(u)$ then  
7: Copy $P_{i}^{j}$ to secondary buffer of $Q_{u}$  
8: $P' \leftarrow P' \cup \{P_{i}^{j}\}$  
9: end if  
10: end for  
11: end for  
12: for all $P_{i}^{j}$ in $Buffer(C)$ do  
13: if $P_{i}^{j}$ contains $flag(u)$ then  
14: Copy $P_{i}^{j}$ to secondary buffer of $Q_{u}$  
15: $P' \leftarrow P' \cup \{P_{i}^{j}\}$  
16: end if  
17: end for  
18: Process packets in secondary buffer  
19: Merge secondary buffer with $Q_{u}$  
20: end if  
21: Process packets in $Q_{u}$  
22: return $P''$

C. Queueing Model

Assuming a Markovian server per switch, queue of each switch is modeled as a $M/M/1/K/\alpha$ queueing system, where incoming packets follow Poisson’s distribution and those packets are processed by the corresponding switch in exponentially distributed service time. Let, $\frac{1}{\lambda_j}$ and $\frac{1}{\mu_j}$ denote the mean service time and mean inter-arrival time at switch $S_j$, respectively. We also consider that each switch has a finite queue length $K$. Figure 2 describes the queueing model for SDN.

Figure 3 shows the state-transition-rate diagram of our proposed queueing model for a single OpenFlow switch. The average packet arrival rate and average service rate for the switch be $\lambda$ and $\mu$, respectively. Therefore, the traffic intensity is $\rho = \frac{\lambda}{\mu}$. The switch is in region $r \in \{high\cupmedium\cuplow\}$. Initially, controller sends update signal to the switch. The switch continues processing until region $r$ starts update. During update of region $r$, the switch queues the received packets unless it completes update. Therefore, the service rate for this stage is $\mu = 0$. If the switch queue is full, the packets are buffered at neighbor queues or at the controller buffer according to Algorithm 2. After the switch completes update, it processes packets from neighbor buffer as well as its own buffer as mentioned in Algorithm 3. Therefore, the new packet arrival rate is $\lambda = \lambda + \lambda'$, where $\lambda'$ is the rate at which packets arrive to the current switch from the buffers of neighbor switches. The traffic intensity in this scenario is $\rho' = \frac{\lambda'}{\mu}$. After the switch processes all the packets stored in neighbor queues, $\lambda'' = 0$ and $\lambda' = \lambda$.

The probability that the switch has $i$ packets, when it receives the update signal from the controller and $r$ is not the current update region is given by:

$$P_{i}^{1} = \rho' P_{0}^{1}, \quad (11)$$

where $P_{0}^{1}$ is the probability that the switch has zero packets when it has received the update signal.

We consider the scenario that region $r$ starts update when the switch has $i$ packets queued and completes update when it has $j$ packets queued. During update, let the probability of having $n \geq i$ packets be $P_{n}^{2}$. We know:

$$P_{i}^{2} = P_{i}^{1} \quad (12)$$

Packets are added to the switch queue at a rate of $\lambda$ and no processing is performed during an ongoing update procedure. Hence, we get:

$$\lambda \times P_{i}^{2} = \lambda \times P_{i+1}^{2} = \ldots = \lambda \times P_{K}^{2} \quad (13)$$

Therefore, we get:

$$P_{i}^{2} = P_{i+1}^{2} = \ldots = P_{K}^{2} = P_{i}^{1} \quad (14)$$

Let $P_{j}^{3}$ be the probability that the switch has $j$ packets and it has completed update. From Equation (14) we get:

$$P_{j}^{3} = P_{j}^{2} = P_{i}^{1} \quad (15)$$

The probability $P_{j}^{3}$ can also be expressed as:
update_sig = yes, region update start = no

\[
P^3 = (\rho')^3 P_0^3,
\]

where \( P_0^3 \) is the probability that the switch has zero packets after it has completed update.

From Equations (11),(15), and (16) we have:

\[
\rho P_0^1 = (\rho')^3 P_0^3
\]

\[
P_0^3 = \frac{\rho}{(\rho')^3} P_0^1
\]

According to queueing theory for finite queue length, at steady state:

\[
P_0^1 + P_1^1 + P_2^1 + \ldots + P^K_1 = 1
\]

From Equations (18) and (19) we have:

\[
P_0^1 = \frac{1 - \rho}{1 - \rho K + 1}
\]

\[
P_0^3 = \frac{1 - \rho'}{1 - (\rho')^K + 1}
\]

Hence, from Equations (18) and (22), the probability \( P_0^1 \) is defined as:

\[
P_0^1 = \frac{(\rho')^3 (1 - \rho)}{\rho (1 - (\rho')^K + 1)}
\]

Let \( L_s \) and \( L_o \) be the expected number of packets in the switch before starting update and after completion of update, respectively. According to definitions of queuing theory, we know:

\[
L_s = \sum_{n=0}^{K} n P_n^1 = \sum_{n=0}^{K} n \rho^n P_0^1
\]

\[
= \frac{\rho}{(1 - \rho)(1 - \rho K + 1)} (1 + K \rho^{K+1} - (K + 1) \rho^K)
\]

\[
L_o = \sum_{n=0}^{K} n P_n^3 = \sum_{n=0}^{K} n (\rho')^n P_0^3
\]

\[
= (1 - \rho') (1 - (\rho')^{K+1}) (1 + K (\rho')^{K+1} - (K + 1) (\rho')^K)
\]

Let \( W_s \) and \( W_s' \) be the mean waiting time at the switch before starting update and after completion of update, respectively. From Little theorem, we know:

\[
W_s = \frac{L_s}{\lambda}
\]

\[
W_s' = \frac{L_s'}{\lambda}
\]

Therefore, the increase in mean waiting time at the OpenFlow switch due to update is given by:

\[
W_s' - W_s = \frac{L_s'}{\lambda} - \frac{L_s}{\lambda}
\]

\[
= \frac{1}{\mu} \left[ \frac{1 + K (\rho')^{K+1} - (K + 1) (\rho')^K}{(1 - \rho')(1 - (\rho')^{K+1})} - \frac{1 + K (\rho')^{K+1} - (K + 1) (\rho')^K}{(1 - \rho)(1 - \rho K + 1)} \right]
\]

After the switch completes processing all the packets stored in neighbor queues, the difference \( W_s' - W_s \) becomes zero, eventually.

From Equations (21) and (23) we have:

\[
\frac{1 - \rho}{1 - \rho K + 1} = \frac{(\rho')^2 (1 - \rho')}{\rho' (1 - (\rho')^{K+1})}
\]

Simplifying Equation (29) we get:

\[
\rho' (\rho - 1) + \rho' (1 - \rho') (\rho')^{K+1} - (1 - \rho K + 1) (\rho')^{j+1} + (1 - \rho K + 1) (\rho')^j = 0
\]

**Theorem 1.** The maximum arrival rate of packets from neighbor queues after completion of update is \( \lambda_{max}'' = \left( \frac{1}{n + 1 + K - \frac{1}{1 - (\rho K + 1)}} - \lambda \right) \) for \( j = K \).
Proof. Putting \( j = K \) in Equation (30), we get:
\[
\rho^i(\rho - 1) + \rho^i(1 - \rho)(\rho')^{K+1} - (1 - \rho^{K+1})(\rho')^K + (1 - \rho^{K+1})(\rho')^K = 0
\]  
(31)

Differentiating both sides of Equation (31) w.r.t. \( \rho' \), we get:
\[
(\rho')^K(\rho^i(1 - \rho)(K + 1) - (1 - \rho^{K+1})(K + 1))
\]
\[
+ (1 - \rho^{K+1})\frac{K}{\rho} = 0
\]
(32)

As \( (\rho')^K \neq 0 \), we get:
\[
\rho^i(1 - \rho)(K + 1) - (1 - \rho^{K+1})(K + 1)
\]
\[
+ (1 - \rho^{K+1})\frac{K}{\rho} = 0
\]
(33)

Based on Equation (33), \( \rho' \) is defined as:
\[
\rho' = \frac{K}{1 + K(1 - \rho)(1 - \rho^{K+1})}
\]
(35)

Differentiating Equation (32) w.r.t. \( \rho' \), we get:
\[
(\rho')^{K-1}(\rho^i(1 - \rho)(K + 1) - (1 - \rho^{K+1})(K + 1))
\]
\[
+ (1 - \rho^{K+1})\frac{K}{\rho} = 0
\]
(36)

Putting value of \( \rho' \) from Equation (35) in Equation (36), we get:
\[
(\rho')^{K-1}(\rho^i(1 - \rho)(K + 1) - (1 - \rho^{K+1})(K + 1))
\]
\[
+ (1 - \rho^{K+1})\frac{K}{\rho} = 0
\]
(37)

As \( 0 < \rho < 1 \), \( (\rho^i - 1) < 0 \) and \( (1 - \rho^{K+1}) < 0 \). Therefore, the expression \((\rho')^{K-1}(\rho^i(1 - \rho)(K + 1) - (1 - \rho^{K+1})(K + 1)) + (1 - \rho^{K+1})\frac{K}{\rho} < 0\). Hence, the maximum value of \( \rho' \) is
\[
\frac{K}{1 + K(1 - \rho)(1 - \rho^{K+1})}.
\]
(38)

Hence, the maximum arrival rate of packets from neighbor queues after completion of update is
\[
\lambda_{max} = \left[ \frac{1}{\mu} \frac{1}{1 + K(1 - \rho)(1 - \rho^{K+1})} \right] - \lambda \text{ for } j = K.
\]

Theorem 2. The minimum arrival rate of packets from neighbor queues after completion of update is
\[
\lambda_{min}'' = \left[ \frac{1}{\mu} \frac{1}{(1 - \rho^{K+1})(1 - \rho)} \right] - \lambda \text{ for } j = 0.
\]

Proof. For \( j = 0 \) we get \( i = 0 \) as \( 0 \leq i \leq K \) and \( j \geq i \). Putting \( j = 0 \) and \( i = 0 \) in Equation (30), we get:
\[
(\rho - 1) + (1 - \rho)(\rho')^{K+1} - (1 - \rho^{K+1})\rho''
\]
\[
+ (1 - \rho^{K+1}) = 0
\]
(39)

Solving Equation (39) \( \rho'' \) is expressed as follows:
\[
\rho'' = \frac{(1 - \rho^{K+1})}{(K + 1)(1 - \rho)}
\]
(40)

Differentiating Equation (39) w.r.t. \( \rho' \), we get:
\[
K(\rho + 1)(1 - \rho)(\rho')^{K-1}
\]
(41)

As \( 0 < \rho < 1 \), \( (1 - \rho) > 0 \). Therefore, the expression \((1 - \rho)(\rho')^{K-1} > 0\). Hence, the minimum value of \( \rho' \) is
\[
\frac{1}{\mu} \frac{1}{(1 - \rho^{K+1})(1 - \rho)} - \lambda \text{ for } j = 0.
\]

IV. PERFORMANCE EVALUATION

In this section, we evaluate the performance of CURE in terms of the following metrics: update duration, average rule space utilization, average packet waiting time, and inconsistent packet count. To evaluate the performance metrics, we have implemented a discrete event simulator in MATLAB and performed two experiments. In the first experiment, we measure update duration and average rule space utilization while varying number of switches in a leaf-spine topology with \( \frac{2N}{3} \) leaf (ingress) switches and \( \frac{N}{3} \) spine switches (e.g., Sprint, NetRail, and Compuverse). We run five test flows in each of these topologies to compute the performance metrics: average packet waiting time and inconsistent packet count.

<table>
<thead>
<tr>
<th>TABLE II: Simulation parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>Number of switches in leaf-spine topology</td>
</tr>
<tr>
<td>Rule space size in a switch</td>
</tr>
<tr>
<td>Upper bound on controller-to-switch delay</td>
</tr>
<tr>
<td>Upper bound on end-to-end network delay</td>
</tr>
<tr>
<td>Upper bound on time interval between dispatch of two consecutive update messages</td>
</tr>
<tr>
<td>Average packet arrival rate per switch</td>
</tr>
<tr>
<td>Average packet service rate per switch</td>
</tr>
<tr>
<td>Average queue size per switch</td>
</tr>
<tr>
<td>Flow table lookup time</td>
</tr>
</tbody>
</table>

A. Simulation Parameters

Table II represents the parameters which we have considered for the simulation. We implemented leaf-spine topology by varying the total number of switches from 6 to 48. The maximum number of flow entries in a switch has been fixed to 8000 [27]. As shown in Table II, we consider that the upper bounds on controller-to-switch delay, end-to-end network delay, and the time interval between generation of two consecutive update messages are 4.865 ms, 0.262 ms, and 5.24 ms, respectively [5]. The average packet arrival rate, average packet service rate, and average queue size per switch are 0.2 million packets per second (mpps) [29], 0.03 mpps [28], and 0.1 – 0.9 million packets, respectively. We consider that the flow table lookup time for each packet is 33.33333 \mu sec [28].

B. Result and Discussion
1) **Update Duration:** Update duration is calculated as the time interval between the dispatch of the first update message by the controller, and update completion of the last switch. Garbage collection, i.e., removal of old rules are included in the update duration, as defined in Definition 6.

2) **Average Rule Space Utilization:** We calculate the average rule space utilization as the percentage of rule space used during different stages of update by $N$ switches in the leaf-spine topology. Figure 6 shows the rule space utilization percentage for two-phase update [12], timed two-phase update [5], and CURE. Rule space utilization is almost similar for two-phase update and timed two-phase update as they both require to store both versions (old and new) of rules until the start of garbage collection phase. Average rule space requirement for CURE is 29.964% and 30.348% less than that of two-phase update and timed two-phase update, respectively. As shown in Figure 6, we synthesize that the average rule space utilization is less in CURE as storage of both version of rules, simultaneously, is not required.

3) **Average Packet Waiting Time:** For each of the three topologies — Sprint, NetRail, and Compuserve, we simulate five test flows, and calculate the average waiting time for the incoming packets that are either waiting in the switch...
queues or are in process. Figure 4 depicts the topologies and the corresponding test flows. We have estimated delay of each link based on the distance between the corresponding nodes. Similar to Ref. [5], we assume 5 microsecond delay per kilometer.

Figures 8a, 8b, and 8c depict the average packet waiting time for different average queue size per switch for each of the test flows in each of the topologies. The average packet arrival rate per switch is 0.02 mpps. We yield that average packet waiting time is almost constant for different queue size per switch for a fixed packet arrival rate. For smaller queue size, incoming packets wait in more number of neighbor queues. Whereas, larger switch queues accommodate more number of incoming packets that are processed sequentially.

Figures 8d, 8e, and 8f depict the average packet waiting time for different average packet arrival rate for each of the test flows in each of the topologies. The average packet queue size is 0.073 million packets. Average packet waiting time increases with increasing packet arrival rate.

4) Inconsistent Packet Count: We measure inconsistency as a percentage of inconsistent packets in the system. Inconsistent packets are identified based on Definition 7.

Figure 9 compares inconsistency count in CURE with the timed two-phase update [5] for different average packet arrival rate. We simulate test flows s1, n1, and c1 in topologies Sprint, NetRail, and Compuserve, respectively. Average queue size per switch is 0.073 million packets. The simulation duration is 5000 milliseconds. In timed two-phase update, inconsistency count decreases with increasing packet arrival rate. For packet arrival rate 0.005 mpps, the average inconsistency percentage for Sprint, NetRail, and Compuserve is 3.937%, 2.172%, and 3.1665%, respectively. For packet arrival rate 0.025 mpps, the average inconsistency percentage for Sprint, NetRail, and Compuserve is 1.478%, 0.607%, and 0.638%, respectively. Whereas, average inconsistency count for CURE is similar for different packet arrival rate. The average inconsistency percentage for Sprint, NetRail, and Compuserve is 0.333%, 0.206%, and 0.245%, respectively. Therefore, we yield that in CURE an initial percentage of incoming packets become inconsistent due to the ongoing network update and inconsistency count reduces as time elapses after completion of the
update.

V. CONCLUSION

This work emphasizes reduction of TCAM usage during SDN update with an aim to increase scalability required for handling big data. This work modifies the update scheme of OpenFlow-enabled SDN and proposes a multilevel queue-based policy for ensuring packet-level consistency. We compare our scheme with other approaches of SDN update to evaluate its performance. Results clearly display enhanced scalability and reduced TCAM usage.

The future work will include extension of the proposed scheme in distributed SDN control plane, where multiple controllers perform network update at the same time. We will consider flow-level consistency along with packet-level consistency.

REFERENCES


