Abstract—In Software Defined Networks, where the network control plane can be programmed by updating switch rules, consistently updating switches is a challenging problem. In a per-packet consistent update (PPC), a packet either matches the new rules added or the old rules to be deleted, throughout the network, but not a combination of both. PPC must be preserved during an update to prevent packet drops and loops, provide waypoint invariance and to apply policies consistently. No algorithm exists today that confines changes required during an update to only the affected switches, yet preserves PPC and does not restrict applicable scenarios. We propose a general update algorithm called PPCU that preserves PPC, is concurrent and provides an all-or-nothing semantics for an update, irrespective of the execution speeds of switches and links, while confining changes to only the affected switches and affected rules. We use data plane time stamps to identify when the switches must move from the old rules to the new rules. For this, we use the powerful programming features provided to the data plane by the emerging programmable switches, which also guarantee line rate. We prove the algorithm, identify its significant parameters and analyze the parameters with respect to other algorithms in the literature.

I. INTRODUCTION

A network is composed of a data plane, which examines packet headers and takes a forwarding decision by matching the forwarding tables in switches, and a control plane, which builds the forwarding tables. A Software Defined Network (SDN) simplifies network management by separating the data and control planes and providing a programmatic interface to the control plane.

SDN applications running on one or more controllers can program the network control plane dynamically by updating the rules of the forwarding tables of switches to alter the behaviour of the network. Every Rules Update consists of a set of updates in a subset $S$ of the switches $B$ in a network. Every update in switch $s_i \in S$ consists of two sets of rules $R_1$ and $R_0$. Rule set $R_0$ is to be deleted and rule set $R_1$ is to be inserted. One of $R_0$ or $R_1$ may be NULL. The rules are denoted by $s_i(R_1, R_0)$. Every switch $s_i, s_i \in S$, is called an affected switch. All other switches ($B - S$) are unaffected switches. The set of rules that is neither inserted nor deleted is an unaffected set of rules and is denoted by $R_u$.

Packets that traverse the network during the course of an update must not be dropped (drop-freedom) and must not loop (loop-freedom) on account of the update. In Figure 1(a), when switches are updated to change a route from the old path (solid lines), to the new path (dotted lines), if $s_f$ is updated first, packets arriving at $s_1$ will get dropped, while if $s_3$ is updated first, packets arriving at $s_3$ will loop. Flows in a network may need to pass through a number of waypoints in a certain order at all times (waypoint invariance). In Figure 1(b), $s_1, s_3$ and $s_5$ are the waypoints to be traversed in that order and this cannot be achieved by sequencing the updates in any order. While the above three properties can be preserved in many update scenarios by finding a suitable order of updates [1, 2, 3, 4], the example in Figure 1(c) cannot be solved by suitably sequencing updates, in any scenario. Initially the network has a policy $P_1$ which requires an ingress switch $s_1$ to drop packets whose ports are in the range $p_1 - p_2$ and $s_5$, an egress switch, to drop packets whose destination addresses are in the range $IP_1$ to $IP_2$. Policies may be distributed over various switches in an SDN to reduce load on middleboxes or on links to them [5] or due to lack of rule space on a switch [6]. When the policy is changed to $P_2$, the network administrator desires either $P_1$ or $P_2$ to be applied to any packet $p$ and not, for example, $P_1$ at $s_1$ and $P_2$ at $s_5$. We call this property pure per-packet consistency (pure PPC).

In order to preserve all the properties for all scenarios, every Rules Update must be per-packet consistent [7] (PPC): every packet $P$ travelling through the network must use either rule $r_0 \in R_0$ in every affected switch $s_i \in S$ or rule $r_1 \in R_1$ in every affected switch $s_i$, and never a combination of both, during a Rules Update. Other rules in the switches are to be used if they match. The number of packets that get subjected to any of the inconsistencies can be quite large because the
time to update a few rules at a switch, which is of the order of milliseconds [9], is far greater than the time to switch a packet out of a switch, which is of the order of nanoseconds, given that the line rate of switches is 10-100 Gbits/s on 10-100 ports [9] and assuming an average packet size of 850 bytes [10]. The basic algorithm that preserves PPC is a two-phase update (2PU) [7]: All the rules in all the switches check for the appropriate version number by all the ingresses. To update a set of rules from \( v_0 \) to \( v_1 \), first the new \( v_1 \) rules and a copy of the unaffected rules are installed to check for \( v_1 \), in every internal (non-ingress) switch. Now all the ingresses start labelling all the packets as \( v_1 \). All \( v_1 \) packets match only the new rules and the \( v_0 \) packets in the network match only the old rules, preserving PPC. After all the \( v_0 \) packets exit the network, the controller deletes the old rules on all the switches. 2PU affects all the switches in the network.

We propose an update algorithm called Proportional Per-packet Consistent Updates, PPCU, that is general, preserves PPC, confines changes to only the affected switches and rules and provides an all-or-nothing semantics for a Rules Update, regardless of the execution speeds of switches and links.

The summary of PPCU is as follows: Let the latest time at which all the switches in \( S \) install the new rules be \( T_{last} \). Each affected switch examines the time stamp, set to the current time by the ingresses, in each data packet. If its value is less than \( T_{last} \), it is switched according to the old rules while if it is greater than or equal to \( T_{last} \), it is switched according to the new rules.

II. MOTIVATION AND RELATED WORK

Why preserving PPC is important: Proper installation of policies require preservation of PPC and no less. Middle box deployment in SDN [5] [11] rely on packets traversing middle boxes in a certain order [12] [13]. Languages [14] that enable SDN application developers to specify and use states, also require the order of traversal of switches that store those states to be preserved during an update. These examples emphasize the importance of preserving waypoint invariance for all scenarios and therefore of PPC. For many algorithms that preserve more relaxed properties, a PPC preserving update is a fall-back option, for example, if they cannot find an order of updates that preserve waypoint invariance [11] or if their Integer Linear Programming algorithms do not converge [15].

Why the number of switches involved in the update must be less: Algorithms that perform a Rules Update while preserving PPC either 1) need to update all the switches \( B \) in the network [16] or 2) identify paths affected by \( S \) and update all the switches along those paths [11] [17] or 3) update only \( S \) and all the ingresses [18] [19], regardless of the number of elements in \( S \) and regardless of the number of rules to be updated.

If paths affected by a Rules Update are identified (category 2), every switch in the path, which may be across the network, needs to be changed, even to modify one rule on one switch. If one rule affects a large number of paths, such as “if TCP port=80, forward to port 1”, computing the paths affected is time consuming [1]. In update methods of categories 1 and 2, the number of rules used in every switch modified during the update doubles, as the new and old rules co-exist for some time. Hence the update time is disproportionately large, even if the number of rules updated is small.

If rules can be proactively installed on switches to prevent a control message being sent to a central controller for every missing rule [20], one controller can manage an entire data centre. In a data centre with a Fat Tree topology with a k-ary tree [21] where the number of ports \( k = 48 \), the number of ingress switches is about 92% of the total number of switches. Even to modify one rule in a switch, all of the ingresses will need to be modified, for algorithms of category 3.

Updating only switches in \( S \) instead of \( S \) and other switches in \( B - S \) will reduce the number of controller-switch messages and the number of individual switch updates that need to be successful for the Rules Update to be successful. We wish to characterize this need as footprint proportionality (FP), which is the ratio of the number of affected switches for a Rules Update to the number of switches actually modified for the update. In the best case the FP is 1, and there is no general update algorithm that achieves this while maintaining PPC, as far as we know.

Why using rules with wildcards is important: Installing wildcarded rules for selected flows or rule compression in TCAM for flows or policies [22] reduces both space occupancy in TCAM [23] and prevents sending a message to the controller for every new flow in the network [20]. [24] provides solutions for the tradeoff between visibility of rules and space occupancy. Thus in a real network, using wildcarded rules is inevitable. Many of the existing solutions do not address rules that cater to more than one flow [8] [11]. A Rules Update may require only removing rules and not adding any rule or vice versa, on one or more switches in a network. Though [19] addresses this, it needs changes to all the ingresses for all the rules for any update and limits concurrency. We wish to have \( FP = 1 \) for all the update scenarios possible.

Why concurrent updates must be supported: Support of multiple tenants on data center networks require updating switches for virtual to physical network mapping [25], [26]. Applications that control the network [27], [28], [29] and VM and virtual network migration [30] will require frequent updates to the network, making large concurrent updates common to SDNs. Existing algorithms that preserve PPC and allow concurrent updates limit the maximum number of concurrent updates [18] [19] [11], with the last one becoming impractical if the number of paths affected by the update is large. 2PU [7] does not support concurrency.

Why algorithms that use data plane time stamping are inadequate: Update algorithms that use data plane time stamping either cater to only certain update scenarios or require very accurate and synchronous time stamping [31] or do not guarantee an all-or-nothing semantics for Rules Updates as they depend on an estimate for the time at which the update must take place [32] [33], thus relying on the execution speeds
Our contributions in this paper are:

1) We specify and prove an update algorithm PPCU that:
   a) requires modifications to only the affected switches \( S \) and
      affected rules, while preserving PPC. b) allows practically
      unlimited concurrent non-conflicting updates. c) allows all
      update scenarios: this includes Rules Updates that involve only
      deletion of rules or only insertion of rules or a combination
      of both and Rules Updates that involve forwarding rules that
      match more than one flow. d) expects the switches to be
      synchronized to a global clock but tolerates inaccuracies. e) updates all of \( S \) or none at all, irrespective of the execution
      speeds of the switches or links.

2) We prove the algorithm, analyze its significant parameters
   and find them to be better than comparable algorithms.

3) We illustrate that the algorithm can be implemented at line
   rate.

III. USING DATA PLANE PROGRAMMABILITY:

With the advent of line rate switches whose parsers,
match fields and actions can be programmed in the field
using languages such as P4 [34] and Domino [9], it has
become possible to write algorithms for the data plane in
software. Programmable switch architectures such as Ban-
zai [9], RMT [55], FlexPipe [49] (used in Intel FM6000 chip
set[37]) and XPliant[38] aid this and languages to develop
SDN applications leveraging these abilities [39][14] and P4
applications [40] [41] indicate acceptability.

Data plane programmability is important to PPCU for
several reasons: 1) It is easier to change protocols to add
and remove headers to and from packets 2) It is possible to
process packet headers in the data plane, still maintaining line
rate 3) Once the packet processing algorithm is specified in
a language, the language compiler finds the best arrangement
of rules on the physical switch [42]. It is possible that rules
that require a ternary match may be stored in TCAM and
exact match rules in SRAM [55] or the former in a Frame
Forwarding Unit and the latter in a hash table [36].

4) The actual match-field values in the rules, the actions,
the parameters associated with actions and the variables associated
with rules can be populated and modified at run time. How
to do this is a part of the control plane and is proposed to be
“Openflow 2.0” [34]. Specifications such as SAI[43] or Thrift
[44] or the run-time API generated by the P4 compiler [45]
[41] may be used at present. We assume that switches support
data plane programmability and can be programmed in P4.

IV. MODEL DESCRIPTION

Description of a programmable switch model: The ab-
stract forwarding model advocated by protocol independent
programmable switches [45] consists of a parser that parses
the packet headers, sends them to a pipeline of ingress match-
action tables (the ingress pipeline) that in turn consist of a set
of match-fields and associated actions, then a queue or a buffer,
followed by a pipeline of egress match-action tables (the egress
pipeline). In this section, we describe only the features of P4
that are relevant to PPCU.

A P4 program consists of definitions of 1) packet header
fields 2) parser functions for the packet headers 3) a series of
match-action tables 4) compound actions, made of a series of
primitive actions and 5) a control flow, which imperatively
specifies the order in which tables must be applied to a
packet. Each match-action table specifies the input fields to
match against; the input fields may contain packet headers
and metadata. The match-action table also contains the actions
to apply, which may use metadata and registers. Metadata is
memory that is specific to each packet, which may be set by
the switch on its own (example: value of ingress port) or by the
actions. It may be used in the match field to match packets or
it may be read in the actions. Upon entering the switch for the
first time, the metadata associated with a packet is initialised
to 0 by default. A register is a stateful resource and it may be
associated with each entry in a table (not with a packet). A
register may be written to and read in actions.

P4 provides a set of primitive actions such as \texttt{modify} \_\texttt{field}
and \texttt{add} \_\texttt{header} and allows passing \texttt{parameters} to these
actions, that may be metadata, packet headers, registers etc.
When the primitive action \texttt{resubmit} is applied to the ingress
pipeline, a packet completes its ingress pipeline and then
resubmits the \texttt{original} packet header and the possibly modified
metadata associated with the packet, to the parser; the metadata
is available for matching. If there are multiple \texttt{resubmit}
actions, the metadata associated with each of them must be
made available to the parser when the packet is resubmitted.
Similarly, when \texttt{recirculate} is applied to a packet in the egress
pipeline, the packet, with its header modifications and metad-
data modifications, if any, is posted to the parser. Conditional
operators are available for use in compound actions to process
expressions (we use if statements in the paper to improve
readability).

While this specifies the \textit{definition} of the programmable
regions of the switch, actual rules (table entries) and the
parameters to be passed to actions need to be specified and
is described below. These need to be populated by an entity
external to this model - the controller, through the switch CPU.
This is facilitated by a run-time API.

Description of Rules and packets: A switch has an ordered
set of rules \( K \), consisting of \( n \) rules \( r_1, r_2, \ldots, r_n \). Each rule
\( r \) has three parts: a priority \( P \), a match field \( M \) and an
action field \( A \) and is represented as \( r = [P, M, A] \). Let
\( r_1 = [P_1, M_1, A_1] \) and \( r_2 = [P_2, M_2, A_2] \), \( r_1 \prec r_2 \) if \( P_1 > P_2 \).
\( M \) must be as per the match field defined in the table and \( A \)
must be one of the actions associated with the table. A packet
\( P \) only has match fields. An incoming packet is matched with
the match fields of the rules in the first table specified in the
control flow of the switch and the action associated with the
highest priority rule that matches it is executed. The packet is
then forwarded to the next match-action table, as specified in
the control flow.

In \( K \), some rules may be dependent on the other. For
example, in Figure 2 the rules \( a_1, a_3, a_4 \) and \( a_{14} \) depend on
each other. The solid arrows show the existing dependencies among rules and the dotted arrows, the dependencies after a Rules Update.

r with suitable subscripts or superscripts denotes an individual rule, while R with suitable subscripts or superscripts denotes a set of rules, with \( r_0 \in R_0, r_1 \in R_1 \) and \( r_n \in R_n \), throughout the paper. A packet that matches any \( r \in R_1 \) in any \( s \in S \) is called a new packet. A packet that matches any \( r \in R_0 \) in any \( s \in S \) is called an old packet and all old and new packets are called affected packets. The first affected switch containing at least one of \( R_1 \) or \( R_0 \) that a packet matches is denoted as \( s_f \).

**Disjoint Rules Updates:** Let \( P_1 \) be the set of all packets that match at least one of \( R_0 \) or \( R_1 \) of a Rules Update \( U_1 \) and let \( P_2 \) be the set of all packets that match at least one of \( R_0 \) or \( R_1 \) of another Rules Update \( U_2 \). \( U_1 \) and \( U_2 \) are said to be disjoint or non-conflicting if \( P_1 \) and \( P_2 \) are disjoint. If \( U_1 \) and \( U_2 \) are disjoint, they can occur concurrently and every packet in the network will be affected by at most one of the updates \( U_1 \) or \( U_2 \). For example, let \( R_1 = \{[5, 1000, forward 10]\}. Let \( s_1(null, R_1) \) be the only update in \( U_1 \). Let \( R_0 = \{[5, 1 \ast \ast, forward 5]\}. Let \( s_1(R_0, null) \) be the only update of \( U_2 \). \( U_1 \) and \( U_2 \) conflict. Let \( R_1' = \{[5, 1111, forward 10]\}. Let \( s_1(null, R_1') \) be the only update in \( U_3 \). \( U_1 \) and \( U_3 \) are disjoint. One Rules Update can consist of one or more disjoint updates. In Figure 2 \( a_3 \) and \( a_7 \) need to be deleted and \( a_2 \), \( a_6 \) and \( a_{10} \) need to be installed. They can all be part of the same Rules Update.

**Relationship between the old and the new rules:** Two ordered sets of rules \( A \) and \( B \) are said to be match-field equivalent if the total set of packets that they are capable of matching are identical. Thus \( A = \{[P_1, 0000, A], [P_2, 0110, A], [P_3, 0111, A]\} \) and \( B = \{[P_1, 0000, B], [P_2, 0100, B], [P_3, 0110, B]\} \) are match-field equivalent.

The set of rules in \( A \) for which no match-field equivalent rules of priority equal to or lower than \( A \) exists in \( B \) is called the special difference of \( A \) and \( B \), denoted \( A - B \). If \( R_1 = \{[P_1, 0000, A_1], [P_2, 00 \ast \ast, A_2]\} \) and \( R_0 = \{[P_2, 0000, B_1]\} \), where \( P_1 > P_2 \), \( R_1 - R_0 = \{[P_2, 00 \ast \ast, A_2]\} \). The set of rules in \( A \) for which no match-field equivalent rules of priority equal to or greater than \( A \) exists in \( B \) is called the inverse special difference of \( A \) and \( B \), denoted \( A \sim B \). If \( R_0 = \{[P_2, 00 \ast \ast, A_2]\} \) and \( R_1 = \{[P_1, 0000, B_1]\} \), where \( P_1 > P_2 \), \( R_0 \sim R_1 = \{[P_2, 00 \ast \ast, A_2]\} \). Either or both of \( R_0 \sim R_1 \) and \( R_1 - R_0 \) may not be \( \phi \) for every affected switch. For example, in Figure 2 \( \{a_7\} \sim \{a_6\} \neq \phi \) and \( a_{10} \) has no corresponding old rule. Thus, both \( R_0 \sim R_1 \) and \( R_1 - R_0 \) are not \( \phi \). This has implications for the update algorithm, as discussed in section V.

**V. CHALLENGES IN A RULES UPDATE**

Providing an FP of 1 is difficult because multiple disjoint updates may be combined into one Rules Update and a switch rule may belong to more than one flow, due to usage of wildcarded rules. Thus for packets belonging to different flows the first affected switch \( s_f \) may be different - that is, there may be more than one \( s_f \) belonging to a Rules Update. Neither the controller nor the switches know the paths that are affected by the update or the \( s_f \) for each path. Due to these reasons, firstly, the controller cannot instruct a specific switch to relabel packets to switch to the new version of rules, as is done in some of the update algorithms. Secondly, all switches must have the same algorithm, regardless of their position in a flow. Moreover, the rules to be deleted and installed and the unaffected rules depend on each other in complex ways, as discussed in [V] causing difficulty in maintaining PPC. The key insight is that all the affected switches must know if the rest of the affected switches and itself are ready to move to the next version, and after that, if a packet crosses its \( s_f \), it must store that fact in its header; the switch examines the \( TS \), the time stamped at the ingress on every packet, to understand the readiness and stores the result in two one-bit fields \( f_{p_1} \) and \( f_{p_2} \) in the packet.

Since the controller-switch network is asynchronous, when a controller sends messages to switches, all switches will not receive them at the same time, or complete processing them and respond to the controller at the same time or they may not respond at all. Solving race conditions arising out of these scenarios is possible if the values of \( f_{p_1} \) and \( f_{p_2} \) which are set by \( s_f \) determine whether they match \( R_0 \) or \( R_1 \) or \( R_{oa} \), regardless of what the states of the switches in its path are. Also, the clocks at the switches may be out of sync to the extent the time synchronising protocol would permit. In the next section, we shall state the algorithm first and then describe the measures taken in them to address the above challenges.

**VI. ALGORITHM FOR CONCURRENT CONSISTENT UPDATES**

**Data plane changes at the ingress and egress switches:** Each packet \( p \) entering the network has a time stamp field and two one bit fields \( f_{p_1} \) and \( f_{p_2} \) added to it and removed from it programmatically, at the ingress and the egress, respectively. All ingress sets \( TS \) to the current time at the switch and \( f_{p_1} \) and \( f_{p_2} \) to 0 for all the packets entering it from outside the network. Altering the value of \( TS \), wherever required in the update algorithm, occurs only after it is thus set.

**Notifications used:** \( f_{Flag} \) indicates if the rule is new (NEW), old (OLD) or unaffected (U). \( f_{p_1} \), when set to 1, indicates that the packet must be switched only according to OLD rules, where they exist and \( f_{p_2} \), when set to 1 indicates that the packet must be switched only according to NEW rules, where they exist. The metadata bit \( f_1 \) when set to 1 for a packet.
indicates that this packet must not be matched by a NEW rule. The metadata bit \( f_2 \) when set to 1 for a packet indicates that this packet must not be matched by an OLD rule.

Algorithm 1 Algorithm at the controller

```plaintext
1: procedure CONTROLLER()
2:     while 1 do
3:         if event = app_message_received then
4:             for each switch \( s \) in \( S \) do
5:                 SEND_COMMIT(\( v, R_0, R_1 \))
6:             end for
7:         else if event = R2C_recd then
8:             \( T_i = \) time received in Ready To Commit
9:         end if
10:     T_last = the largest value of \( T_i \) received
11:     send CommitOK(\( v, T_{last} \)) to all switches in \( S \)
12: end if
13: else if event = Ack_COK_recd then
14:     \( T_a = \) time received in Ack Commit OK
15:     if Ack COK is received from all affected switches then
16:         \( T_{del} = \) the largest value of \( T_a \) received
17:         send_DiscardOld(\( v, T_{del} \)) to all affected switches
18:     end if
19: else if event = Discard_Old_Ack_recd then
20:     if Discard Old Ack is received from all affected switches then
21:         \( \triangleright \) The update is complete.
22:     end if
23: end if
24: end while
25: end procedure
```

Algorithm 2 Algorithm at the switch

```plaintext
1: procedure SWITCH(null)
2:     while 1 do
3:         if event = commit_recd then
4:             \( \triangleright \) \( R_1 \) and \( R_0 \) are received in Commit
5:             \( \triangleright \) Begin atomic actions
6:             for each rule \( r \) of \( R_0 \) do
7:                 Modify \( f_2 \) and \( f_{p2} \) of \( r \) to check for 0
8:                 Set flag of \( r \) = OLD
9:             end for
10:             for each rule \( r \) of \( R_1 \) do
11:                 Install \( r \) \( \triangleright \) \( f_1 \) and \( f_{p1} \) check for 0
12:                 Set flag of \( r \) = NEW
13:             end for
14:             \( \triangleright \) End atomic actions
15:             \( T_i = \) current time
16:             Send ReadyToCommit(\( v, T_i \))
17:         else if event = commit_OK_recd then
18:             \( T_{last} = \) value of time received in Commit OK
19:             \( \triangleright \) Begin atomic actions
20:             for each rule \( r \) of \( R_0 \) do
21:                 Set \( T \) associated with \( r = T_{last} \)
22:             end for
23:             for each rule \( r \) of \( R_1 \) do
24:                 Set \( T \) associated with \( r = T_{last} \)
25:             end for
26:             \( \triangleright \) End atomic actions
27:             \( T_i = \) current time
28:             Send AckCommitOK(\( v, T_i \))
29:         else if event = discard_old_recd then
30:             \( T_{del} = \) value received in Discard Old
31:             \( T_i = \) current time
32:             \( M = \) maximum time taken for a packet to be removed from the network
33:             START_TIMER(\( T_{del} + M - T_i, v \))
34:             Send DiscardOldAck(\( v \))
35:         else if event = timer_expiry then
36:             Delete \( R_0 \) belonging to \( v \) from the switch
37:             for each rule \( r \) of \( R_1 \) do
38:                 Modify \( f_1, f_2, f_{p1}, f_{p2} \) of \( r \) to check for *
39:                 Set flag = \( U \)
40:             end for
41:             \( \triangleright \) The update is complete. After \( M \) units of time, another update conflicting with \( v \) may begin.
42:         end if
43:     end while
44: end procedure
```

**Algorithm at the control plane:** This section first specifies the algorithm for the control plane, at the controller (Algorithm 1), and at each \( s_i \in S \) (Algorithm 2) and later for the data plane, in Algorithm 3. The message exchanges below are the same as in [16] and [18]; the parameters in them and actions upon receiving them have been modified to suit PPCU.

1) The Controller receives an Update Request from the application, with the list of affected switches \( S, R_0 \) and \( R_1 \), for every switch \( s \in S \). The actions of all rules are as in Algorithm 3. An update identifier \( v \) is associated with each update. The controller sends a “Commit” message to every switch \( s \in S \) (line 5 of Algorithm 1) with \( v, R_0 \) and \( R_1 \) as its parameters.

2) A switch \( s \in S \) receives the “Commit” message, extracts \( v, R_0 \) and \( R_1 \). The switch 1) changes the match part of the installed \( R_0 \) rules to check if \( f_2 = 0 \) and \( f_{p2} = 0 \) 2) installs the \( R_1 \) rules and 3) sets flag = OLD for the \( R_0 \) rules and flag = NEW for the \( R_1 \) rules. The above actions must be atomic, as indicated in the algorithm. Now it sends “Ready to Commit” with the current time of the switch \( T_i \) as a parameter (lines 2 to 16 in Algorithm 2).

\(^1\)It is assumed that the controller and the switch maintain all parameters related to an update in a store and check whether each message received is appropriate; we omit these details and the error handling required for an all-or-nothing semantics to improve clarity.
3) The controller, upon receiving “Ready To Commit”, stores the current time received. Let $T_{last}$ be the largest value of time received in “Ready To Commit” (Algorithm 1 line 10). Now it sends “Commit OK” to all switches $s \in S$ with $T_{last}$ (Algorithm 1 line 11).

4) Upon receiving “Commit OK”, the switch sets $T = T_{last}$ in $R_0$ and $R_1$, which is considered a single atomic action. It now sends the current time $T_1$ in “Ack Commit OK”. See lines 7 to 28 in Algorithm 2.

5) The controller receives “Ack Commit OK” from all the switches. Let the largest value of time received in “Ack Commit OK” be $T_{del}$. It sends “Discard Old” to all the switches $s$ with $T_{del}$.

6) Let the time at which a switch $s$ receives “Discard Old” be $T_s$. The switch sends “Discard Old Ack” to the controller. The switch starts a timer whose value is $T_{del} + M - T_s$ where $M$ is the maximum lifetime of a packet within the network and $T_{del}$ the time received in “Discard Old”. (line 33 in Algorithm 2). When the timer expires, all the packets that were switched using the rules $R_0$ are no longer in the network. Therefore the switch deletes $R_0$. It sets $flag = U$ for every $R_1$ rule and modifies its $f_1$, $f_2$, $f_{p1}$ and $f_{p2}$ to check for $\ast$, as shown in lines 33 to 40 in Algorithm 2. With this, the update at the switch is complete.

7) After the controller receives “Discard Old Ack” messages from all the switches, the update is complete (line 21 in Algorithm 1) at the controller.

Note: After $M$ units after timer expiry at the last affected switch, the last of packets tagged $f_{p2} = 1$ will exit the network. Now the next update not disjoint with the current one may begin.

Algorithm at the data plane - actions at switches:
Algorithm 3 specifies the template for a compound action associated with every rule in a match-action table in a switch. Each rule has two metadata fields of 1 bit each, $f_1$ and $f_2$, associated with it, initialised to 0 by default, indicating that the packet is entering that table in that switch for the first time (If other tables in the same switch has rule updates, a separate set of two bits need to be used for each of those tables. For ease of exposition, we describe only one table being updated). Each rule has two registers, $T$, initialised to $T_{max}$, which is 1 less than the maximum value that $TS$ can have, and $flag$, which decides if the rule is old (OLD), new (NEW) or unaffected (U), initialised to $U$. Unaffected rules have $f_1$, $f_2$, $f_{p1}$ and $f_{p2}$ set to $\ast$ in their match-fields and do not check for the value of $TS$. A new rule is always installed with a priority higher than that of an old rule. In P4, rules with ternary matches have priorities associated with them as such rules can have overlapping entries.

A rule may execute any of its own actions, which is what is referred to in [7][18] and [31] in Algorithm 3. The parameters $action\_params$ that are passed to the compound action are in turn passed to the actions associated with the rule. For example, a new rule may require a packet to be forwarded to port 5, instead of port 6. Then, the table entry for the new rule would pass 5 as a parameter from its rule, while the old rule would pass 6.

Suppose $s_f$, the first affected switch of affected packet $p_1$ in Rules Update $U_1$, receives “Commit”. $TS < T_{max}$ of $p_1$ and let $R_1 - R_0 = \phi$ for $s_f$. $p_1$ will match $r_1$ in $s_f$, get its $f_2$ set to 1 and get resubmitted (line 12). Now it will not match $r_1$ and instead match $r_0$, get its $f_{p1}$ set to 1 (line 17) and get switched by $r_0$ in subsequent affected switches and $r_u$, if no $r_0$ exists. After $s_f$ receives “Commit OK” and sets its $T = T_{last}$ when a packet $p_2$ whose $T > T_{last}$ arrives at $s_f$, it gets switched by the new rule, gets its $f_{p2}$ set to 1.
and gets switched by $r_1$ rules in all the subsequent affected switches and $r_u$ if no $r_1$ exists. Suppose in Rules Update $U_2$, $s_f$ has $R_1 - R_0 \neq \phi$ and receives “Commit”. Now a packet $p_3$ entering it with $TS < T_{max}$ gets its $f_1$ set to 1, and then resubmitted (line 12). The resubmitted packet matches another rule $r_u$ and gets its $f_{p1}$ set to 1 (line 27), making the packet use $r_0$ rules in subsequent switches and $r_u$, where no $r_0$ exists. After $s_f$ receives “Commit OK”, a packet $p_4$ whose $T > T_{last}$ matches $r_1$, gets its $f_{p2}$ set to 1 and gets switched in subsequent switches by $r_1$ or $r_{u}$, if $r_1$ does not exist. The case for an update where $s_f$ has $R_0 \sim R_1 \neq \phi$ is similar.

**Specific scenarios:** We shall use a list (inexhaustive) of scenarios to illustrate that each step in Algorithm 3 is necessary: Consider two switches $i < S$ and $i \in S$, in Figures 3 and 4 far apart from each other in the network. In all the cases, the lower priority rule forwards a packet to a port while a higher priority rule forwards a packet to the same port and increments a field $F$ in the packet. The value of $TS$ and the rest of the packet header are shown in boxes with dotted lines, before and after crossing $s_f$. The numbers 1, 2 and 3 against the boxes indicate the sequence of occurrence of events.

1) **Case 1. new rules must not be immediately applied:** After receiving “Commit”, if any $s_i$ installs $r_1$ earlier than others, as packets will immediately begin to match the new rules, PPC will be violated. Therefore, each rule $r_1 \in R_1$ must check if $TS \geq T$ where $T = T_{max}$ before it executes the actions associated with $r_1$ (line 5).

2) **Case 2, when to switch to the new rules: $T$ must not be $T_{max}$** for packets to ever start using $r_1$. That time must be after all the affected switches have installed the new rules and is now ready to use them, which is $T_{last}$. Therefore after receiving “Commit OK”, each affected switch will set $T = T_{last}$ using the run time API and start using the new rules if $TS \geq T$ of an incoming packet. However, this is insufficient to guarantee PPC.

3) **Case 3. $s_f$ receives “Commit OK” first:** See Figure 3. Suppose a packet $p$ whose $TS > T_{last}$ arrives at the first affected switch $s_f$. Since $s_f$ has received “Commit OK”, it switches the packet using $r_0$ rules. Now the next affected switch $s_j$ has not received “Commit OK” yet. Therefore $r_1$ in $s_j$ checks if $TS \geq T_{max}$ and since that is not so, $p$ gets switched using $r_0$ rules, thus violating PPC. To solve this problem, $r_1$ of $s_f$ must set $f_{p2} = 1$ of all incoming packets that match $r_1$ and whose $TS \geq T$. $r_1$ must check if $f_{p2} = 1$ in the action (lines 5 and 6).

4) **Case 4, $s_f$ receives “Commit OK” last:** To solve this problem, illustrated in Figure 3, $r_0$ must set $f_{p1} = 1$ and check for $f_{p1} = 1$ in the action (lines 16 and 17). $r_1$ must check for $f_{p2} = 0$ in its match field, so that $r_1$ rules in subsequent affected switches do not match this packet and therefore always match $r_0$.

5) **Case 5, $R_1 - R_0 \neq \phi$ for $s_f$ and $s_j$:*** To solve this issue, illustrated in Figure 4, the switch must set $f_{p1} = 1$ for packets whose $TS < T$ and matching $R_1 - R_0$ (lines 9 to 12 and line 27). For this, the switch first sets $f_1 = 1$ and then resubmits the packet. Since $r_1$ checks for $f_1 = 0$, the packet does not match $r_1$. No $r_0$ exists for this packet. Therefore the packet matches $r_u$, as intended. In $r_u$, if $f_1 = 0$, $f_{p1}$ is set to 1.

6) **Case 6, $R_0 \sim R_1 \neq \phi$ for $s_j$:*** To prevent a PPC violation, in Figure 4, all old rules $r_0 \in R_0$ must check if $f_{p2} = 0$ in its match field (line 14). Thus if a packet whose $TS \geq T_{last}$ arrives, it will not use any rule $r_0 \in R_0$, instead it will use the next matching rule.

7) **Case 7, $R_0 \sim R_1 \neq \phi$ for $s_f$:*** See Figure 4. $r_0$ gets deleted from $s_f$ after its timer expires (line 35) in Algorithm 2. The solution for the issue here is to set $f_{p2} = 1$ of packets that match $r_0$ as well, if their $TS \geq T_{last}$ (lines 21 to 23 and line 29). Packets with $f_{p2} = 1$ will use $r_1$ rules if they match and unaffected rules if they do not, as unaffected rules do not check for $f_{p1}$ or $f_{p2}$. Similar to the explanation in section VI-3 such a packet needs to be resubmitted after setting $f_2 = 1$ and $r_0$ must check for $f_2 = 0$. Subsequently, in $r_0$, the packet has its $f_{p2}$ set to 1. Since metadata instances are always initialised to 0 (45), unaffected rules need not set $f_1$ and $f_2$ to 0.

On resubmitting packets: It must be noted that packets need to be resubmitted 1) only at $s_f$ and 2) only after receiving “Commit OK” and until the timer expires, if $s_f$ has $R_0 \sim R_1 \neq \phi$ for an update. In all other cases, packets need
to be resubmitted 1) only at $s_f$ and 2) only after receiving “Commit” and until the switch receives “Commit OK”. The action recirculate must be used instead of resubmit, if the change is to a table in the egress pipeline.

**Impact of multiple tables:** The algorithm requires no changes to support updates to more than one ingress or egress table in a switch. The update to each table must be atomic with respect to the update to another table within the same switch. For example, when a switch receives “Commit OK”, it must set $T = T_{last}$ to the affected rules in all the tables, for that Rules Update, atomically. How a switch implementation addresses this is outside the scope of this paper. [46] is a line of work in this direction.

**VII. Concurrent Updates**

Each disjoint Rules Update requires a unique update identifier $v$ for the duration of the update, to track the update states at the affected switches and the controller. The number of disjoint Rule Updates that can be simultaneously executed is limited only by the size of $v$. $v$ is exchanged only between the controller and the switches and hence is not dependent on the size of a field in any data packet. Therefore as many disjoint updates as the size of the update identifier or the processing power of switches would allow can be executed concurrently.

**VIII. Using time stamps**

**Feasibility of adding TS at the ingress:** We assume that all switches have their clocks synchronized and the maximum time drift $\gamma$ of switches from each other is known. If the network supports Precision Time Protocol, $\gamma = 1 \mu$sec [31]. Intel FM6000, a programmable SDN capable switch, supports PTP, its $\gamma < 1 \mu$sec and the time stamp is accessible in software [37]. The size of the register used to store a packet timestamp in FM6000 is 31 bits. P4 supports a feature called “intrinsic metadata”, that has target specific semantics and that may be used to access the packet timestamp. Using the add_header and remove_header actions and intrinsic metadata, the $TS$ field may be added at the ingress and removed at the egress for every packet, for targets that support protocols such as PTP.

**Asynchronous time at each switch:** Since a single rule update in a TCAM is of the order of milliseconds, a time stamp granularity of milliseconds is sufficient for Rules Updates. Let us assume that the timestamp value is in milliseconds and that an ingress $s_i$ is faster than an affected internal switch (at the most by $1 \mu$sec). Let $t_1$ be the time stamp of a packet $p$ in the network that is stamped by $s_i$, even before the Rules Update begins. After the Rules Update begins, let the (temporally) last affected switch send its current time stamp $T_{last}$ in “Ready To Commit”. Let $t_1 > T_{last}$, since the ingress is faster. Now a switch $s_j \in S$, which is not an $s_f$, will switch $p$ with the new rules, violating PPC. To prevent this, each switch may set $T = [T_{last} + \gamma + 1]$, instead of $T = T_{last}$. If an ingress clock is slow, there will be no PPC violations. Thus PPCU tolerates known inaccuracies in time synchronisation. To reduce packet transmission times, the size of $TS$ may be reduced, thus reducing the granularity. $s_f$ will need to wait longer to receive packets whose $TS \geq T_{last}$, thus lengthening the Rules Update time. Since the size of $TS$ is programmable in the field, the operator may be choose it according to the nature of the network.

**IX. Proof**

We need to prove that the algorithms[1,2,3] together provide PPC updates. $p(m, f_p1, f_p2)$ denotes an affected packet with match-field $m$ and fields $f_p1$ and $f_p2$. $r(m)$ denotes a rule whose match-field is $m$ (in both, $m$ excludes $TS$, $f_p1$ and $f_p2$). By definition of $S$, for a packet $p(m, f_p1, f_p2)$, any $s \in S$ has either a rule $r_0(m)$ or $r_1(m)$ or both.

Let us assume that the individual algorithms[1,2,3] are correct. Let us assume that switches are synchronized, to make descriptions easier. In the description below, when we refer to a packet, we mean a packet affected by the update. The update duration is split into three intervals: before $s_f$ receiving “Commit”, from $s_f$ receiving “Commit” to the timer expiring, and after the timer expiry.

**Property 1:** After the Rules Update begins and before $s_f$ receives “Commit”, PPC is preserved: $s_f$ (and all $s_j$ that have not received “Commit”) switches all packets using a rule that will be marked $r_0(m)$ on receiving “Commit” or $r_1(m)$ (if no rule is to be deleted on that switch). The first $s_j$ that receives “Commit” switches using $r_0(m)$ or $r_1(m)$ (the latter if the switch does not have $r_0(m)$ but has only $r_1(m)$) and sets $f_p1 = 1$ for all received packets. Subsequent $s_j$s will not use $r_1(m)$ as $f_p1 = 1$ and use $r_0(m)$ if it exists (or $r_1(m)$ if no $r_0(m)$ exists). Thus packets follow $r_0(m)$ if it exists or $r_1(m)$ if $r_0(m)$ does not exist, preserving PPC.

**Property 2:** Between $s_f$ receiving “Commit” at $s_f$ and its timer expires PPC is preserved: Property 2.1: As per Algorithms[2,3] if $s_f$ has received “Commit”, $p(m, f_p1, f_p2)$ will exit $s_f$ at either $(p(m, 0, 1)$ or $(p(m, 1, 0))$, until its timer expires.

Property 2.2: $s_f$ will receive any of the types of packets 1) $p(m, 0, 0)$ with its $TS < T_{last}$ or 2) $p(m, 1, 0)$ or 3) $p(m, 0, 1)$ and no other type of packet, until its timer expires. Proof: Packets of type 1 may reach $s_f$ as it may have crossed $s_f$ before $s_f$ received “Commit” and $TS$ will always be less than $T_{last}$ of such packets, as $T_{last}$ is the time at which the last switch has received “Commit”. If a packet of type other than these three reaches $s_j$, it means it has not crossed $s_f$, by Property 2.1. In that case, the only possibility is that $s_f$ is the first switch, which is not true by definition.

By Algorithm[3] packets of types 1 and 2 referred to in Property 2.2 can use only $r_0$ rules (or $r_1$ if $r_0$ does not exist) on all switches $s_j$. By the same algorithm, packets of type 3 match only $r_1$ rules (or $r_0$ if $r_1$ does not exist) on all switches $s_j$. In $s \notin S$, packets match $r_{uw}$, $r_u$ can change the value of $f_p1$ or $f_p2$ of a packet only if $f_1$ or $f_2$ for that packet is set, which occurs only on an affected switch. Therefore, PPC is preserved.

**Property 3:** When the timer expires at any switch $s_i \in S$, all packets that were switched using $r_0$ at least at one $s \in S$ have left the network: Let us consider two cases:
1) All packets whose $TS < T_{last}$ get switched by $s_f$ using $r_0$ rules, by Algorithm 3. Such packets exit the network by time $T_{last} + M$. However, $T_{del} > T_{last}$. 2) All packets whose $TS \geq T_{last}$ that arrive at $s_f$ get switched using $r_0$, until that $s_f$ receives its “Commit OK”, by Algorithm 3. At time $T_{del}$, the last of $s_f$ have received “Commit OK” and have started switching packets using $r_1$. Hence at time $T_{del} + M$, all the packets (whether due to case 1 or 2) that were switched using $r_0$ have left the network. The current time at a switch is $T_1$ and the switch receives an instruction to start a timer that expires $T_{del} + M$, giving the timer a value of $M - T_1 + T_{del}$. Therefore when this timer expires, all the old packets have left the network.

Property 4: After timer expiry at $s_f$: All packets in the network have their $TS \geq T_{last}$ after timer expiry at $s_f$, as all packets whose $TS < T_{last}$ have left the network by the timer expiry at any switch, by Property 3. Therefore all packets leaving $s_f$ are of the form $p(m, 0, 1)$. Therefore all switches $s_j \neq s_f$ will use $r_1$ if it exists (or $r_0$ if $r_1$ does not exist), if the timer has not expired on $s_j$. If the timer has expired on $s_j$, it will only cause it to not check for $f_{p1}$, $f_{p2}$, $f_1$ or $f_2$, which makes no difference to matching the rule that the packet matches. Since packets have only $r_1$ to match (or $r_u$ where it does not exist), PPC is preserved.

Thus due to Properties 1 to 4, PPC is preserved during and after the update.

X. Analysis of the Algorithm

The symbols used in the analysis are; $\delta$: the propagation time between the controller and a switch, $t_i$: the time taken to insert rules in a switch TCAM, $t_d$: the time taken to delete rules from the TCAM, $t_m$: the time taken to modify rules in the TCAM, $t_o$: the time taken to modify registers associated with a rule, $t_u$: the time for which a switch waits after it receives “Discard Old” and before it deletes rules, $n_v$: the number of old rules that need to be removed, $n_a$: the number of new rules that need to be added, $n$: the maximum number of rules in a switch, $k_o$: the number of affected switches, $k_i$: the number of ingresses, $k_t$: the total number of switches, $R_1$: The time between the switch receiving “Commit” and sending “Ready To Commit”, $R_2$: The time between the switch receiving “Commit OK” and sending “Ack Commit OK” and $R_3$: The time between the switch receiving “Discard Old” and the switch performing its functions after timer expiry. It is assumed that the value of $\delta$ is uniform for all switches and all rounds, the values of time are the worst for that round, the number of rules, the highest for that round and that the processing time at the controller is negligible. Since unaffected rules have ternary matches for $f_{p1}$ and $f_{p2}$, we assume that all the match fields are stored in TCAM and the corresponding actions in SRAM [35][42], for PPCU, and for other algorithms.

We add to and evaluate using the parameters of interest identified for a PPC in [16]: 1) Overpass: Duration for which the old and new rules exist at each type of switch 2) Transition time: Duration within which new rules become usable from the beginning of the update at the controller 3) Message complexity: the number of messages required to complete the protocol 4) Time complexity: the total update time 5) Footprint Proportionality 6) Concurrency: Number of disjoint concurrent updates.

The purpose of the analysis is to understand what the above depend upon and to compare with a single update in E2PU (using rule updates in TCAMS) [16], which updates switches using 2PU with 3 rounds of message exchanges, and with the algorithm in [19]. We assume that [19] uses acknowledgements for each message sent. Since all three have similar rounds, it is meaningful to compare the time taken by each round as given in table I. The Overlap is $\delta + R_1 + R_2 + R_3$ and Transition time is $3\delta + R_1 + R_2$ for all the algorithms under consideration.

Since the values associated with the action part of a rule ($T$ and $f$) are stored in SRAM, we assume that update times of these values (in ns) will be negligible compared to TCAM update times and ignore the terms that involve $t_o$. We find in table I that 1) PPCU has lower message complexity, assuming $k_o > k_a/2$ and better FP 2) PPCU and [19] have comparable times for Rounds 1 and 3. For Round 2, PPCU fares better. Therefore, PPCU fares better for Overlap, Transition time and Time complexity. 3) PPCU has better concurrency. From [VI-7] it may be inferred that for PPCU, packets need to be resubmitted either for a duration of $R_2 + 2\delta + R_3$ or $R_1 + 2\delta + R_2$ at only $s_f$. Since resubmit is an action supported by line rate switches, we assume that the delay due to a resubmission at $s_f$ for this time frame during an update is tolerable.

Feasibility of implementation at line rate: Adding and removing headers such as $TS$, $f_{p1}$ and $f_{p2}$ are feasible at line rate; so are setting and checking metadata, such as $f_1$ and $f_2$ and header fields $f_{p1}$ and $f_{p2}$ in actions, as per table 1 in RMT [35]. While compiling the action part, Domino [9] checks if operations on stateful variables in actions can run

| Parameter | PPCU | E2PU [16] | [19] |
|-----------|------|----------|------|
| Message complexity | $6k_o$ | $4k_t$ | $4k_a + 4k_i$ |
| TP | $1$ | $k_o/k_t$ | $k_o/(k_o + k_i)$ |
| Round 1 (R1) | $n_o(t_m + t_o) + n_n(t_i + t_o)$ | $(n - n_o + n_n) t_i + (n_o + n_n)^t_m$ | $(n_o \times t_m) + (n_n \times t_i)$ |
| Round 2 (R2) | $(n_o + n_n) t_o$ | $(n - n_o + n_n) t_m + n_n t_i$ | $n \times t_m$ |
| Round 3 (R3) | $t_o + n_o (t_m + t_o) + n_n t_d$ | $t_o + n \times t_d$ | $t_o + n_o (t_m + n_n t_d)$ |
| Round 4 | Not applicable | $n \times t_m$ |
| Propagation Time P | $\delta$ | $\delta$ | $\delta$ |
| Time Complexity | $P + R1 + R2 + R3$ | $P + R1 + R2 + R3$ | $P + R1 + R2 + R3 + R4$ |
| Concurrency | Unlimited | $0$ | Number of bits in version field |

Excludes the time for current packets to be removed from the network at the end of the update, where applicable.
at line rate by mapping those operations to its instruction set - PPCU requires only reading the state variables flag and T and this can be achieved using the “Read/Write atom” (name of instruction) in Domino. This demonstrates the feasibility of PPCU running at line rate.

XI. CONCLUSIONS

PPCU is able to provide non-conflicting per-packet consistent updates with an all-or-nothing semantics, with updates confined to the affected switches and rules, with no restrictions on the scenarios supported and no limits to concurrency. The algorithm fares better than other algorithms that preserve PPC, in a theoretical evaluation of significant parameters. PPCU algorithm fares better than other algorithms that preserve PPCU, on the scenarios supported and no limits to concurrency. The confined to the affected switches and rules, with no restrictions on the maximum time drift of switches from each other is known, which is the case with PTP.

REFERENCES

[1] W. Zhou, D. Jin, J. Croft, M. Caesar, and B. P. Godfrey, “Enforcing customizable consistency properties in software-defined networks,” in NSDI 2015, May.
[2] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, “Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies,” in Proceedings of the 15th ACM Workshop on Hot Topics in Networks. ACM, 2014.
[3] S. Dudycz, A. Ludwig, and S. Schmid, “Can’t touch this: Consistent network updates for multiple policies,” in Proc. 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2016.
[4] D. M. F. Mattos, O. C. M. B. Duarte, and G. Pujolle, “Reverse update: A consistent policy update scheme for software defined networking,” Communications Letters, IEEE, vol. 20, no. 5, May 2016.
[5] Y. Ben-Izhak, K. Barabash, R. Cohen, A. Levin, and E. Raichstein, “EnforSDN: Network policies enforcement with SDN,” in 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM).
[6] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, “Rules placement problem in openflow networks: a survey,” IEEE Communications Surveys & Tutorials, 2016, to appear.
[7] M. Reitblatt, N. Foster, J. Rixford, C. Schlesinger, and D. Walker, “Abstractions for network update,” in SIGCOMM 2012.
[8] X. Jin, H. H. Liu, R. Gandhi, S. Kundula, R. Mahajan, M. Zhang, J. Rixford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in SIGCOMM 2014.
[9] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown, “Packet transactions: High-level programming for line-rate switches,” in “SIGCOMM 2016, to appear.
[10] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” ACM SIGCOMM Computer Communication Review, vol. 40, no. 1, 2010.
[11] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, “OpenNF: Enabling innovation in network function control,” ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, 2015.
[12] D. A. Joseph, A. Tavakoli, and I. Stoica, “A policy-aware switching layer for data centers,” ACM SIGCOMM Computer Communication Review, vol. 38, no. 4, 2008.
[13] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, “Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions,” in Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, 2013.

[14] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, “SNAP: Stateful network-wide abstractions for packet processing,” SIGCOMM 2016, to appear.
[15] L. C. Stefano Vissicchio, “FLIP the (flow) table: Fast lightweight policy-preserving SDN updates,” in INFOCOMM 2016. IEEE.
[16] R. Sukuparam and G. Barua, “Enhanced algorithms for consistent network updates,” in IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN). IEEE, 2015.
[17] N. P. Katta, J. Rexford, and D. Walker, “Incremental consistent updates,” in Proceedings of the second ACM SIGCOMM workshop on Hot topics in Software Defined Networking 2013. ACM.
[18] R. Sukuparam and G. Barua, “CCU: Algorithm for Concurrent Consistent Updates for a Software Defined Network,” in Twenty Second National Conference on Communications : NCC. IEEE, 2016.
[19] S. Luo, H. Yu, and L. Li, “Consistency is not easy: How to use two-phase update for wildcard rules?” Communications Letters, IEEE, vol. 9, no. 3, 2015.
[20] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, “On scalability of software-defined networking,” Communications magazine, IEEE, vol. 51, no. 2, 2013.
[21] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” SIGCOMM Comput. Commun. Rev., vol. 38, no. 4, Aug. 2008.
[22] S. Zhang, F. Ivanic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik, “An adaptable platform rule for software-defined networks,” in 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE.
[23] S. Luo, H. Yu et al., “Fast incremental flow table aggregation in SDN,” in 2014 23rd International Conference on Computer Communication and Networks (ICCCN). IEEE, 2014.
[24] A. S. Iyer, V. Mann, and N. R. Samineni, “Switchreduce: Reducing switch state and controller involvement in openflow networks,” in IFIP Networking Conference, 2015. IEEE.
[25] M. Banikazemi, D. Olsheski, A. Shaiikh, J. Tracey, and G. Wang, “Meridian: an SDN platform for cloud network services,” Communications Magazine, IEEE, vol. 51, no. 2, 2013.
[26] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshive, G. Parulkar, E. Salvadori, and B. Snow, “OpenVirtex: Make your virtual SDNs programmable,” in Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014.
[27] T. Benson, A. Anand, A. Akella, and M. Zhang, “MicroTE: Fine grained traffic engineering for data centers,” in CoNEXT. ACM, 2011.
[28] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks,” in NSDI, vol. 10, 2010.
[29] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, “ElasticTree: Saving energy in data center networks.” in NSDI, vol. 10, 2010.
[30] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, N. McKeown, and M. Zhang, “Fast incremental flow table aggregation in SDN,” in Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.
[31] T. Mizrahi and Y. Moses, “Software defined networks: Its about time,” in SIGCOMM. ACM, 2011.
[32] T. Mizrahi, E. Saat, and Y. Moses, “Timed consistent network updates,” in INFOCOM, 2016.
[33] T. Mizrahi, E. Saat, and Y. Moses, “Timed consistent network updates,” in INFOCOM, 2016.
[34] “Openflow switch specification version 1.5.0,” 2014.
[35] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rixford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al., “P4: Programming protocol-independent packet processors,” ACM SIGCOMM Computer Communication Review, vol. 44, no. 3, 2014.
[36] R. Ozdag, “White paper:intel® Ethernet Switch FM6000 Series-Programmable match-action processing in hardware for SDN,” in Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014.
[37] [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-switch-fm5000-fm6000-datasheet.pdf
[38] [Online]. Available: http://www.cavium.com/XPlant-Ethernet-Switch-Product-Family.html
[39] C. Schlesinger, M. Greenberg, and D. Walker, “Concurrent NetCore: From policies to pipelines,” ACM SIGPLAN Notices, vol. 49, no. 9, 2014.
[40] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, “HULA: Scalable load balancing using programmable data planes,” in SOSR, 2016.
[41] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, “DC.p4: programming the forwarding plane of a data-center switch,” in SOSR, 2015.
[42] L. Jose, L. Yan, G. Varghese, and N. McKeown, “Compiling packet programs to reconfigurable switches.” in NSDI, 2015.
[43] [Online]. Available: http://www.opencompute.org/wiki/Networking/SpecsAndDesigns#Switch_Abstraction_Interface
[44] “Apache thrift.” [Online]. Available: https://thrift.apache.org/
[45] “The P4 language specification, version 1.1.0, January 27, 2016.”
[46] J. H. Han, P. Mundkur, C. Rotsos, G. Antichi, N. H. Dave, A. W. Moore, and P. G. Neumann, “Blueswitch: Enabling provably consistent configuration of network switches,” in Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems. IEEE Computer Society, 2015, pp. 17–27.