Update Algebra: Toward Continuous, Non-Blocking Composition of Network Updates in SDN

Geng Li*†, Y. Richard Yang*, Franck Le‡, Yeon-sup Lim‡, Junqi Wang§
*Yale University, USA, †Tongji University, China, ‡IBM T.J. Watson Research Center, USA, §Rutgers University, USA

Abstract—**The ability to support continuous network configuration updates is an important ability for enabling Software Defined Networks (SDN) to handle frequent or bursty changes. Current solutions for updating SDN configurations focus on one single update at a time, leading to slow, sequential (*i.e.*, blocking) update execution. In this paper, we develop update algebra, a novel, systematic, theoretical framework based on abstract algebra, to enable continuous, non-blocking, fast composition of multiple updates. Specifically, by modeling each data-plane operation in the set of data-plane operations to be executed by an update as a *set-theoretical projection*, update algebra defines novel *operation composition* so that the number of projections for the same match remains constant regardless of the number of updates to be composed, leading to substantial performance benefits. Specifying the dependencies of the data-plane operations in updates as a subset of a *free monoid* in the general case and as partial ordering for *basic consistency*, update algebra defines *update composition* that preserves consistency, even under partially-executed updates, to guarantee correctness. We conduct asymptotic analysis, extensive benchmarking using a real controller, and integration with a real application to demonstrate the benefits of update algebra. In particular, our asymptotic analysis demonstrates that in independent-update dominant settings, update completion time of update algebra remains asymptotically constant despite growth of the number of updates to be executed. Our benchmarking shows that update algebra can achieve 16x reduction in update latency even in settings with an update arrival rate of only 1.6/s. Our integration with Hedera, a real SDN traffic engineering application, shows that update algebra can reduce average link bandwidth utilization by 30% compared with sequential updates.**

I. Introduction

The ability to provide continuous, rapid, non-blocking network configuration updates is an essential capability for Software Defined Networking (SDN). First, it provides a foundation for the development of advanced applications with frequent network updates, which are typically prohibited or discouraged in current SDN systems. A recent trend is the application of machine learning to continuously and rapidly adapt routing strategies to minimize maximum link utilization, achieve proportional fairness, or maximize other objectives [1]–[3]. In addition, with studies having revealed that traffic in several settings (*e.g.*, data centers) is highly dynamic, many solutions are advocating switching traffic at a finer granularity than flows, including subflows, and flowlets, to reduce congestion, and optimize path choices more frequently [4]–[6]. These trends and applications emphasize the need for controllers to support continuous, rapid, non-blocking network configuration updates. Second, a network may experience a set of rapid bursts of changes, causing an SDN controller to receive and handle a batch of network configuration updates [7]. For example, such updates may stem from the occurrence of unpredictable events including outages, Denial of Service attacks, BGP re-routes, or flash crowds.

Although a large amount of research effort has recently been devoted to developing efficient algorithms to update forwarding rules in SDN, existing solutions are unsatisfactory, and do not provide the required capability to handle frequent or short sudden groups of network changes. This is because despite having developed algorithms reduce the numbers of changes, or minimize the network update completion times [8], or preserve a range of properties [9]–[12] including loop and blackhole freedom, existing solutions focus on one single network configuration update at a time. In other words, they execute consecutive received network updates individually, and sequentially in a blocking manner [13]. Similar to the development of pipelining executions of instructions that has led to fundamental changes and performance improvements in computer architecture, the ability to execute continuous and non-blocking updates can lead to significant potential improvements in SDN control architecture [14].

Achieving continuous, non-blocking network updates is not straightforward. The first challenge is to guarantee correctness; a naïve execution may lead to unnecessary blocking due to dependencies on updates. To illustrate this, consider two consecutive updates: a first update U_1 sets the route for a flow, and a following update U_2 changes part of the route. Therefore, in a straightforward execution, the execution of U_2 cannot start until U_1 is finished, leading to a sequential (*i.e.*, blocking execution) update model. The second challenge is that a network configuration update often consists of multiple operations that must be executed at different switches in a specific order to guarantee properties, such as blackhole freedom and waypoint routing (*e.g.*, to traverse a sequence of VNFs) [12]. When executing consecutive updates in a non-blocking manner, these properties must also be preserved. The third challenge is that because network configuration updates often consist of multiple operations, updates do not operate atomically, and when a new update arrives, previous updates may be mid-execution. The execution status may even be unknown to the controller due to the fundamentally distributed nature of the SDN system.

In this paper, we develop a novel, systematic, and foundational theoretical framework based on abstract algebra to reason about and support continuous, non-blocking updates. The framework is motivated by the following two insights. First, when handling multiple updates (*i.e.*, multiple batches of operations), operations on the same flow rules from consecutive updates may be replaced by fewer equivalent ones. For example, the creation of a flow rule followed by its modification can be replaced by the creation of an updated rule. To realize this insight, we model each operation as a *set-theoretical projection*, which provides flexible composition between operations. As such, the first challenge can be adequately addressed. Second, an update can be represented by a set of feasible sequences of operations whose order ensures the desired properties, and composition of multiple updates can be modeled as the application of different mathematical operations on these sequences. In abstract algebra, such a model can be well defined by a *free monoid* (of which a typical example is words with letters). By modeling each update as a subset of a *free monoid* on a set of projections, we can take advantage of the algebraic properties (*i.e.*, associativity, idempotence, selectivity, commutativity) of the structure to identify equivalent operation sequences that preserve correctness and consistency properties, whether the former updates have been completely or partially executed. This insight addresses the second and third challenges.

We conduct asymptotic analysis, extensive benchmarking using a real controller, and integration with a real application to demonstrate the benefits of update algebra. The asymptotic analysis demonstrates that in independent-update dominant settings, the completion time with the existing sequential execution grows linearly, while that of the update algebra remains asymptotically constant. The benchmarking results show that update algebra can achieve 16x reduction in update latency even in settings with an update arrival rate of only 1.6/s Finally, the integration with a real application shows that by applying update algebra, SDN Traffic Engineering applications (*e.g.*, Hedera [2]) can reduce the average link bandwidth utilization by 30% compared to sequential execution.

II. Basic Models

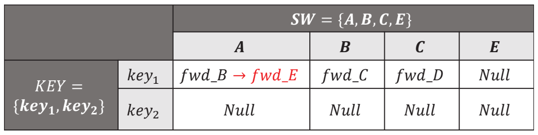
This section introduces the basic models to represent the network data plane configuration and individual network updates. The update algebra framework for the continuous, non-blocking composition of consecutive network updates will be specified in Section III. TABLE I summarizes the main variables and notations used in the following content.

TABLE I: Terminologies

$SW = \{sw\}$	the set of switches (forwarding tables)
$M = \{m\}$	the set of all possible match values
$PR = \{pr\}$	the set of priorities
$key = (key.m, key.pr)$	flow rule key, defined by a match value endowed with a priority
$KEY = \{key\} = M \times PR$	the set of 2-tuple flow rule keys
$AC = \{ac\}$	the set of all possible actions
$AC^+ = AC \cup \{Null\}$	the action set with a null value
$C : SW \times KEY \to AC^+$	data plane configuration
$o(sw, key, ac) : C \to C'$	data plane operation
$u = o_1 o_2 \ldots o_k$	update representative
$U = \{u_1, u_2, \ldots\} : C \to C'$	data plane update
$O(U)$	constituent operations of U
$\Omega(U)$	order constraint of update U

A. Data Plane Configurations

An SDN is comprised of a set of switches SW. A data plane configuration C consists of a collection of flow rules that determine the packets' forwarding states in the network. A flow rule defines an action $ac \in AC$ for flows matching a $key \in KEY$ at switch $sw \in SW$. A key therefore has two attributes: (1) the matching criteria ($key.m$), and (2) a priority ($key.pr$). Equation (1) illustrates four flow rule keys.



		$SW = \{A, B, C, E\}$			
		A	B	C	E
$KEY = \{key_1, key_2\}$	key_1	$fwd\_B \to fwd\_E$	fwd_C	fwd_D	$Null$
	key_2	$Null$	$Null$	$Null$	$Null$

Fig. 1: Illustration of a data plane configuration C with four switches and two keys. A data plane operation $o(A, key_1, fwd\_E) : C \to C'$ is applied on C to get C' where the changed value is labeled in red.

The matching criteria $key.m$ can have wildcards (*) to match ranges of values, and $key.pr$ is set to a finite integer number where a higher number means a higher priority: if a flow matches the matching criteria from multiple keys, the one with the highest priority is preferred and selected.

$$\begin{aligned} key_1 &= (pr = 1, m = \{src\_ip = 10.0.0.1\}),\\ key_2 &= (pr = 1, m = \{src\_ip = 10.0.0.2\}),\\ key_3 &= (pr = 2, m = \{dst\_ip = 10.0.0.*\}),\\ key_4 &= (pr = 2, m = \{src\_ip = 10.0.1.1, dst\_port = 22\}). \end{aligned} \tag{1}$$

An action ac of a flow rule represents the instruction that is applied to the flows matching it. An action can be forwarding to a specified next-hop, modifying a packet, or pipeline processing. The proposed models support the concept of multi-table pipelines in a switch: each flow table can simply be represented as an individual virtual sw. Formally, we define a data plane configuration as follows.

Definition 1 (Data Plane Configuration)**.** *A* ***data plane configuration*** *C of a network is defined as a map from the set of switches and flow rule keys to the set of actions $C : SW \times KEY \to AC^+$, where $AC^+ = AC \cup \{Null\}$.*

Therefore, a configuration C can be expressed with a 2-dimensional matrix over $SW \times KEY$ where each element $C_{sw,key}$ is an action $ac \in AC^+$ for (sw, key). $C_{sw,key} = Null$ represents the absence of a flow rule with key key at sw. Fig. 1 illustrates an example of C with four switches and two keys.

B. Data Plane Updates

A data plane update U consisting of a set of data plane operations $O(U)$ on multiple switches and flow rules can change one configuration to another. Data plane operations act on keys at a particular switch and fall into one of three categories: addition, modification, or deletion. We denote these operations as $\{add(sw, key, ac), mod(sw, key, ac), del(sw, key)\}$; *e.g.*, $add(sw, key, ac)$ means to add an action ac for key at sw. For example, consider the network topology depicted in Fig. 2. In the first update U_1, flows from source IP address 10.0.0.1 (key_1) are forwarded along the route $A \to B \to C \to D$; in the second update U_2, flows from source IP address 10.0.0.2 (key_2) are forwarded along $B \to C \to D$, and the forwarding path for key_1 is changed to $A \to E \to C \to D$ (*e.g.*, to balance the network load). Fig. 2 illustrates the corresponding data plane operations in both U_1 and U_2.

For further derivation, we introduce a more general expression o, parameterized by $SW \times KEY \times AC^+$: add, mod and del are all special cases of o; *e.g.*, $add(sw, key, ac)$ or $mod(sw, key, ac)$ can be expressed as $o(sw, key, ac)$, and $del(sw, key)$ as $o(sw, key, Null)$. An operation o transforms an arbitrary configuration C to another C' as follows:

Definition 2 (Data Plane Operation)**.** *A* data plane operation *$o(sw, key, ac)$ is defined as a* ***morphism*** *between two configurations,* i.e.*, $o : C \to C'$ or $C \xrightarrow{o} C'$, using:*

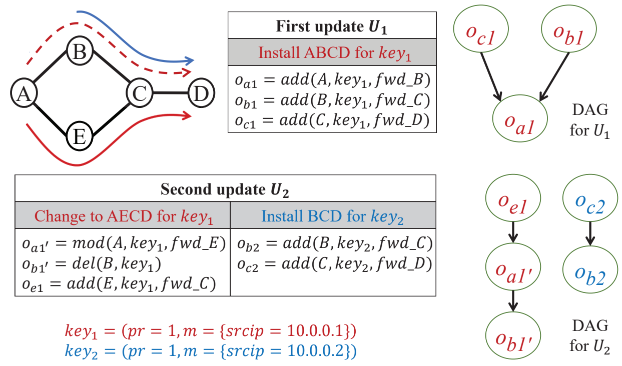

A
B
C
D
E
First update U_1
Install ABCD for key_1
$o_{a1} = add(A, key_1, fwd\_B)$
$o_{b1} = add(B, key_1, fwd\_C)$
$o_{c1} = add(C, key_1, fwd\_D)$
o_{c1}
o_{b1}
o_{a1}
DAG for U_1
Second update U_2
Change to AECD for key_1
Install BCD for key_2
$o_{a1'} = mod(A, key_1, fwd\_E)$
$o_{b1'} = del(B, key_1)$
$o_{e1} = add(E, key_1, fwd\_C)$
$o_{b2} = add(B, key_2, fwd\_C)$
$o_{c2} = add(C, key_2, fwd\_D)$
o_{e1}
o_{c2}
$o_{a1'}$
o_{b2}
$o_{b1'}$
DAG for U_2
$key_1 = (pr = 1, m = \{srcip = 10.0.0.1\})$
$key_2 = (pr = 1, m = \{srcip = 10.0.0.2\})$

Fig. 2: An example of two consecutive updates involving common flows (match key_1). The order constraint ensuring blackhole freedom for each update is illustrated as a DAG of data plane operations.

$$C'_{sw',key'} = [o(sw, key, ac)(C)]_{sw',key'} \tag{2}$$

$$= \begin{cases} ac & \text{if } sw = sw', key = key', \\ C_{sw',key'} & \text{otherwise.} \end{cases} \tag{3}$$

That is, $o(sw, key, ac)$ maps C to C' by changing $C'_{sw,key} = ac$ but preserving other values of C. Fig. 1 shows an example of applying the operation $o_{a1'} = o(A, key_1, fwd\_E)$ on configuration C to obtain C', where the changed value is labeled in red.

Due to the distributed nature of the data plane, operations in an update can be applied in any order, resulting in different intermediate configuration states. However, some intermediate configurations during an update may violate consistency properties such as blackhole/loop-freedom; an *order constraint* is required to specify the feasible operation orders in an update.

Definition 3 (Order Constraint)**.** *The **order constraint** of update U is defined as $\Omega(U)$, which specifies the feasible operation orders in U to ensure consistency properties.*

A concrete instantiation of $\Omega(U)$ will be given in Section III-C. Fig. 2 presents an example of the order constraint to avoid blackholes where no matched rule exists during the updates; the order constraint is represented in the form of a directed acyclic graph (DAG); *e.g.*, the DAG for U_1 shows that o_{a1} must be applied after o_{b1} and o_{c1}. Note that generating the order constraints for various consistency properties is well studied in literature [9]–[12], and therefore not the focus of our work.

Issues with Multiple Updates. The model for individual updates is simple and well documented, but new issues arise when attempting to compose and execute multiple consecutive updates in a non-blocking manner. First, a naïve parallel execution of consecutive updates could lead to non-deterministic or incorrect outcomes. For example, in the scenario of Fig. 2, the execution of U_1, and U_2, could lead to non-deterministic configurations. This is because the operations o_{a1}, and $o_{a1'}$, apply to the same key_1 at switch A, but differ in actions: $o_{a1}.(sw, key) = o_{a1'}.(sw, key)$, but $o_{a1}.ac \neq o_{a1'}.ac$. Consequently, depending on the order in which they were applied, the two operations would lead to different configurations. Further, the order constraints in different updates may have dependencies that affect the non-blocking composition of consecutive updates and may be difficult to identify. For example, $o_{a1'}$ in U_2 must be applied after o_{c1} in U_1 to guarantee the absence of blackholes. Lastly, when a new update arrives, previous updates may be mid-execution. The execution status may even be unknown to the controller due to the fundamentally distributed nature of the SDN system. For example, if U_1 is partially executed when U_2 arrives, the exact execution status may be unknown to the controller.

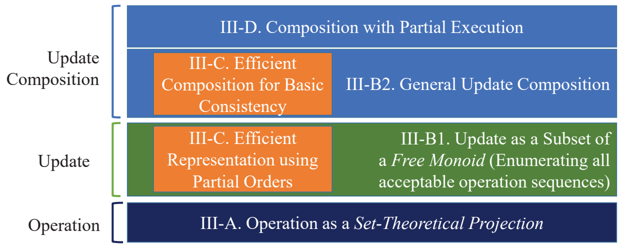

Update Composition
Update
Operation
III-D. Composition with Partial Execution
III-C. Efficient Composition for Basic Consistency
III-B2. General Update Composition
III-C. Efficient Representation using Partial Orders
III-B1. Update as a Subset of a *Free Monoid* (Enumerating all acceptable operation sequences)
III-A. Operation as a *Set-Theoretical Projection*

Fig. 3: Roadmap of update algebra.

III. Update Algebra Framework

In this section, we present our update algebra framework based on advanced abstract algebra. Specifically, data plane operations and updates are modeled by the notions of *set-theoretical projection* (Section III-A) and *free monoid* (Section III-B1), which provide the foundation and substantial algebraic properties for further composition. Based on these models, Section III-B2 proposes a general solution to compose consecutive updates. The solution is general as it can preserve any consistency property. Then, Section III-C introduces an efficient representation and composition using partial order to guarantee specific but commonly required properties. Lastly, Section III-D addresses the issue of composition with partially-executed updates to guarantee correctness under uncertainty. The roadmap of update algebra is illustrated in Fig. 3.

A. Operation as a Set-theoretical Projection

As the basic unit of an update, operation and its composition are first introduced in update algebra, providing the foundation and freedom to replace and rearrange the data plane operations within one update or across distinct updates.

Recall Definition 2 where a data plane operation is defined as a morphism from one configuration to another with one value changed. The operation morphism can be viewed as a *set-theoretical projection* as follows:

Definition 4 (Operation Projection)**.** *Considering the set of all possible configurations over a fixed $SW \times KEY$, such a set is the Cartesian product $(AC^+)^{SW \times KEY}$. Then the operation $o(sw, key, ac)$ can be considered a **projection** from $(AC^+)^{SW \times KEY}$ to the subset $\{C | C_{sw,key} = ac\}$.*

Definition 4 helps us to visualize an operation as a morphism of the configuration space, *i.e.*, the Cartesian product $(AC^+)^{SW \times KEY}$ as shown in Fig. 4. For example, assume $|SW \times KEY| = 2$, then the space is two-dimensional. Therefore, $o(sw, key, ac)$ projects any configuration points onto a line with (sw, key)-component $= ac$, and the operation on the other $(sw*, key*)$ is "orthogonal" to $o(sw, key, ac)$.

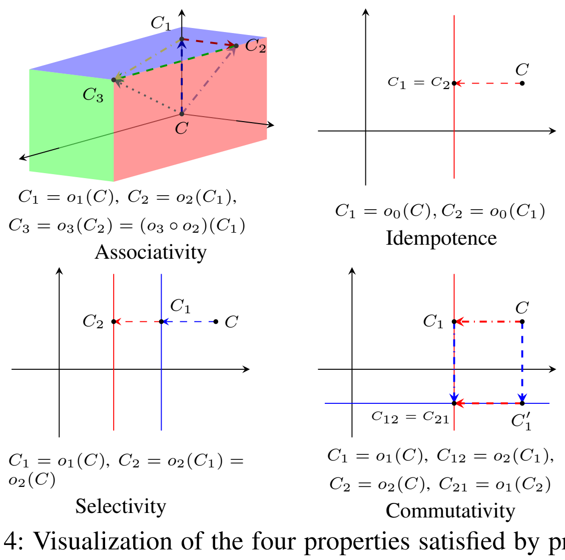
Definition 5 (Morphism Equality)**.** *Two morphisms π_1 and π_2 are **equivalent** iff for any configuration C, $\pi_1(C) = \pi_2(C)$.*

Since both the domain and codomain of an operation morphism are the configuration space, a set of data plane operations $\{o_1, o_2, ...\}$ can be "composed", *i.e.*, arranged in a sequence to form a new morphism. Formally, we define a binary operation $\circ$, called composition of morphisms, such that for any $o_1 : C_0 \rightarrow C_1$ and $o_2 : C_1 \rightarrow C_2$, we have $o_2 \circ o_1 : C_0 \rightarrow C_2$. By modeling an operation as a *set-theoretical projection* based on Definition 4, the operation composition holds the following properties:

Theorem 1 (Operation Composition Properties)**.** *The universal operation set $\mathcal{O} = \{o_1, o_2, ...\}$ and the binary operator $\circ$ have the following four properties under Definition 5.*

1) **Idempotence** (general): $\forall o_1$
$$o_1 \circ o_1 = o_1. \tag{4}$$
2) **Associativity** (general): $\forall o_1, o_2, o_3$
$$(o_3 \circ o_2) \circ o_1 = o_3 \circ (o_2 \circ o_1). \tag{5}$$
3) **Selectivity** (conditional): if $o_1.(sw, key) = o_2.(sw, key)$,
$$o_2 \circ o_1 = o_2. \tag{6}$$
4) **Commutativity** (conditional): if $o_1.(sw, key) \neq o_2.(sw, key)$,
$$o_2 \circ o_1 = o_1 \circ o_2. \tag{7}$$

Here $o_1.(sw, key) = o_2.(sw, key)$ means both the sw and the key of o_1 and o_2 are equal. Theorem 1 can be directly proven by Definition 2. Intuitively, if each morphism is considered as a projection based on Definition 4, we can visualize the four operation properties in the form of projections in a configuration space as in Fig. 4.



C_1
C_2
C_3
C
$C_1 = o_1(C)$, $C_2 = o_2(C_1)$,
$C_3 = o_3(C_2) = (o_3 \circ o_2)(C_1)$
Associativity
$C_1 = C_2$
C
$C_1 = o_0(C)$, $C_2 = o_0(C_1)$
Idempotence
C_2
C_1
C
$C_1 = o_1(C)$, $C_2 = o_2(C_1) = o_2(C)$
Selectivity
C_1
C
$C_{12} = C_{21}$
C_1'
$C_1 = o_1(C)$, $C_{12} = o_2(C_1)$,
$C_2 = o_2(C)$, $C_{21} = o_1(C_2)$
Commutativity

Fig. 4: Visualization of the four properties satisfied by projections.

The properties of real-world SDN implementation (*e.g.*, Flow table modification messages in OpenFlow) align exactly with these properties except for **Selectivity**. TABLE II illustrates a concrete example of **Selectivity**, where $o_1.(sw, key) = o_2.(sw, key)$. Note that if o_2 is a mod, all results after composition become an add. Because in our framework $add(sw, key, ac_2) = mod(sw, key, ac_2) = o(sw, key, ac_2)$, and an add can act as a potent mod (an add operation can override an action even though an action for the identical key already resides in the requested table sw), therefore **Selectivity** property can still hold. Note that $o.key$ is endowed with a determined priority, so operations on overlapping flow rules can be distinguished in composition.

TABLE II: Composition rules for $o_2 \circ o_1$.

$o_2 \setminus o_1$	$\boldsymbol{add(ac_1)}$	$\boldsymbol{mod(ac_1)}$	$\boldsymbol{del()}$
$\boldsymbol{add(ac_2)}$	$add(ac_2)$	$add(ac_2)$	$add(ac_2)$
$\boldsymbol{mod(ac_2)}$	$add(ac_2)$	$add(ac_2)$	$add(ac_2)$
$\boldsymbol{del()}$	$del()$	$del()$	$del()$

B. General Update Representation and Composition

1) Update as a Subset of a Free Monoid: Given the concept of operation composition, an update can be considered as the composition of data plane operations in different sequences, *e.g.*, $o_1 \circ o_2 \circ o_3$ and $o_3 \circ o_1 \circ o_2$. Therefore, we extend the definition of an update using a *free monoid* [15] to address possible operation sequences and capture the order constraint. The formal definitions of the *monoid* and the *free monoid* are given as follows:

Definition 6 (Monoid)**.** *A **monoid** (sometimes called a semi-group with identity element) is a 3-tuple $(S, e, *)$, where S is a set, $e \in S$ is an element, and $*$ is a binary operation $S \times S \rightarrow S$ such that for all $x, y, z \in S$, $x*(y*z) = (x*y)*z$ and $e * x = x * e = x$.*

Definition 7 (Free Monoid)**.** *A **free monoid** S^* on a generating set S is a monoid whose elements are all finite sequences (or strings) of zero or more elements from S, with string concatenation as the monoid operation $*$.*

Example: Letter and Words - A typical *free monoid* example is about letters and words. Start with an alphabet S of letters, $S = \{a, b, c, ..., z\}$. A word on the generating set S is a finite sequence of letters, *e.g.*, $infocom$, and $paris$. Thus, S^* is the set of all possible words, the identity element e is an empty word, and the operation $*$ is word-concatenation. In this *free monoid*, any words can be simply composed together to get a new word, *e.g.*, $no * on = noon$.

Consider a *free monoid* O^*, in which the generating set is a data plane operation set O and the identity element e is an empty update $\varnothing$ (*i.e.*, applying nothing on a configuration C). Then an update can be modeled as follows:

Definition 8 (Update Representation)**.** *An **update** is represented as a set $U = \{u_1, u_2, ...\}$ where u_i, called a **representative**, is a sequence of elements from $O(U)$, and satisfies the order constraint $\Omega(U)$ and the following conditions:*

- *Constitution: $\forall o \in O(U)$, $o \in u_i$;*
- *Distinction: $\forall o_i, o_j \in u_i$, $o_i.(sw, key) \neq o_j.(sw, key)$.*

Remark. U is a subset of the *free monoid* $O(U)^*$ on the generating set $O(U)$. Constitution condition guarantees that all representatives have the constituent operations. Distinction condition avoids configuring a flow rule twice in an update. Each representative u_i representing an order to compose $O(U)$ can transform a C to another as follows:

Definition 9 (Update Representative Morphism)**.** *An update representative $u_i = o_1 o_2 ... o_k$ can be considered as a mor-*

$$\begin{tikzcd}
C \arrow[r, bend left=40, "u_1 = o_1 \circ o_2 \circ \cdots \circ o_n"] \arrow[r, "u_2 = o_2 \circ o_1 \circ \cdots \circ o_n"] \arrow[r, bend right=40, "\vdots \; u_i \in U"'] & C'
\end{tikzcd}$$

(a) Equivalence of representatives.

$$\begin{tikzcd}
C_0 \arrow[r, "U_1"] \arrow[dr, "U_2 \circ U_1"'] & C_1 \arrow[d, "U_2"] \\
& C_2
\end{tikzcd}$$

(b) Update composition.

Fig. 5: Diagrams of the update algebra. (a) Applying any representative of an update achieves the same result. (b) If $U_1 : C_0 \rightarrow C_1$ and $U_2 : C_1 \rightarrow C_2$, then $U_2 \circ U_1 : C_0 \rightarrow C_2$.

phism between two configurations, i.e., $u_i : C \rightarrow C'$, *where* $C' = u_i(C) = (o_1 \circ o_2 \circ ... \circ o_k)(C)$.

Theorem 2 (Equivalence of Representatives). *Given an update* U*, for any configuration* C*, we have* $u_i(C) = u_j(C)$, $\forall u_i, u_j \in U$.

Theorem 2 can be simply proven with **Commutativity** in Theorem 1 and the conditions in Definition 8. Fig. 5(a) illustrates the diagram of Theorem 2. According to the *monoid* presentation theory [15], an update U is an equivalence class of representatives in a *free monoid*, thus can be further defined as follows:

Definition 10 (Update Morphism). *An update* U *can be considered a morphism between two configurations,* i.e., $U : C \rightarrow C'$, *where* $C' = U(C) = u_i(C)$, $\forall u_i \in U$.

Definitions 8 and 10 illustrate an update in the perspectives of representation and morphism respectively. Updates as morphisms are equipped with a composition operation. The diagram of the update composition is illustrated in Fig. 5(b), and such composition presents the following properties:

Theorem 3 (Update Composition Properties). *The universal update set* $\mathcal{U} = \{U_1, U_2, ...\}$ *and the binary operator* $\circ$ *have* ***Associativity*** *and* ***Idempotence*** *properties under Definition 5;* i.e., $\forall U_1, U_2, U_3$*, we have* $(U_3 \circ U_2) \circ U_1 = U_3 \circ (U_2 \circ U_1)$ *and* $U_1 \circ U_1 = U_1$.

Associativity in Theorem 3 is inherited from a *monoid* as in Definition 6, and **Idempotence** can be proven by **Idempotence** and **Commutativity** of operation composition in Theorem 1.

2) General Update Composition: Represented by a subset of $O(^*U)$ enumerating all acceptable operation sequences, an update can be determined by its constituent operations $O(U)$ and order constraint $\Omega(U)$. Therefore, the composition $U_k \circ ... \circ U_1$ is determined by $O(U_k \circ ... \circ U_1)$ and $\Omega(U_k \circ ... \circ U_1)$.
Computation of $\boldsymbol{O(U_k \circ ... \circ U_1)}$. The goal of update composition is to obtain the equivalent operation sequences with a lower operation number. Based on Associativity in Theorem 1, multiple updates can be concurrently composed in a flexible way. Given a set of updates $\{U_1, U_2, ..., U_k\}$, we propose a general solution to compute $O(U_k \circ ... \circ U_1)$ with the following steps:

1) Choose an arbitrary representative u_i from each update U_i, $i \in [1, k]$,
2) Concatenate them with $\circ$, *i.e.*, $u_k \circ ... \circ u_1$,
3) Simplify the concatenated sequence of operations using the properties in Theorem 1 until Distinction condition in Definition 8 is satisfied,

Example. Consider the composition of U_1 and U_2 in Fig. 2, *i.e.*, $U_2 \circ U_1$. Randomly choose their representatives as u_1 and u_2 respectively, and then the composition $u_2 \circ u_1$ can be simplified as follows:

$$\begin{aligned}
u_2 \circ u_1 &= (o_{a1'} o_{b1'} o_{e1} o_{b2} o_{c2}) \circ (o_{a1} o_{b1} o_{c1}) \\
&= (o_{a1'} \circ o_{b1'} \circ o_{e1} \circ o_{b2} \circ o_{c2}) \circ (o_{a1} \circ o_{b1} \circ o_{c1}) \\
&= o_{a1'} \circ o_{a1} \circ o_{b1'} \circ o_{b1} \circ o_{e1} \circ o_{b2} \circ o_{c2} \circ o_{c1} \\
&\qquad \text{by Associativity and Commutativity in Theorem 1} \\
&= o_{a1'} \circ o_{b1'} \circ o_{e1} \circ o_{b2} \circ o_{c2} \circ o_{c1} \\
&\qquad \text{by Selectivity Theorem 1} \\
&= o_{a1''} \circ o_{b1'} \circ o_{e1} \circ o_{b2} \circ o_{c2} \circ o_{c1} \\
&\qquad o_{a1'} \text{ is changed to } o_{a1''} \text{ according to TABLE II}
\end{aligned}$$

Computation of $\boldsymbol{\Omega(U_k \circ ... \circ U_1)}$. To obtain all acceptable update sequences (representatives) from $O(U_k \circ ... \circ U_1)$, we also need $\Omega(U_k \circ ... \circ U_1)$, so that the consistency properties can be preserved. Given a set of operations, existing work on consistent updates provide a large number of efficient algorithms to generate an order constraint for various consistency properties, including loop-freedom [9], and waypoint routing [11]. A general solution is to take $O(U_k \circ ... \circ U_1)$ as an input and run one of the algorithms to get $\Omega(U_k \circ ... \circ U_1)$. For properties whose order constraint consists of partial orders, Section III-C below presents a solution to compute $\Omega(U_k \circ ... \circ U_1)$ more efficiently.

C. Efficient Representation and Composition for Basic Consistency

The update representation using a subset of a *free monoid* is complete, but the sequence set U can get large, especially when composing updates with long sequences of operations; for example, if U_1 has n_1 sequences and U_2 has n_2, a naïve concatenation could result in $n_1 \times n_2$ possible sequences. This section introduces an efficient and compact way to specify and derive the order constraints consisting of partial orders. This format of order constraint guarantees the basic consistency property defined in Definition 11 below.

Definition 11 (Basic Consistency). *The basic consistency property is defined as blackhole- and loop-freedom.*

Basic consistency is commonly used in networks to avoid packet drops and traffic loops [9], [16]. To ensure it, the update execution can be represented by a DAG as in Fig. 6. Consider each edge in the DAG to be a partial order pair, the $\Omega(U)$ for the basic consistency can be represented by a strict partial order set, whose elements are of the form $o_1 \prec o_2$, denoting that o_1 ought to be applied before o_2. The generation of the DAG and partial order set for the composition can be found in [17]. For example in Fig. 6, $\Omega(U_1) = \{o_{c1} \prec o_{a1}, o_{b1} \prec o_{a1}\}$ and $\Omega(U_2) = \{o_{e1} \prec o_{a1'}, o_{a1'} \prec o_{b1'}, o_{c2} \prec o_{b2}\}$. The strict partial order $\prec$ satisfies the following properties:

1) $o \nprec o$,
2) if $o_1 \prec o_2$ and $o_2 \prec o_3$, then $o_1 \prec o_3$,
3) if $o_1 \prec o_2$, then $o_2 \nprec o_1$.

Based on the partial ordering, we propose an efficient solution to achieve update composition, *e.g.*, $U_k \circ ... \circ U_1$. In our solution, the constituent operations $O(U_k \circ ... \circ U_1)$ can be obtained by the same solution as in Section III-B2, while $\Omega(U_k \circ ... \circ U_1)$ is computed as follows:

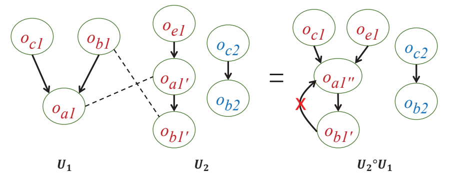


Fig. 6: An example of the efficient composition of U_1 and U_2 in Fig. 2 to preserve the basic consistency, $o_{a1''} = o_{a1'} \circ o_{a1}$ is computed according to TABLE II.

- Step 1: Combine the order constraints together, *i.e.*, $\Omega(U_k \circ ... \circ U_1) = \Omega(U_1) \cup \Omega(U_2) \cup ... \cup \Omega(U_k)$;
- Step 2: Replace operations by composed counterparts in $O(U_k \circ ... \circ U_1)$;
- Step 3: Search cycles in $\Omega(U_k \circ ... \circ U_1)$ based on the properties of $\prec$;
- Step 4.1: Break each cycle by removing the order element inherited from earlier updates, *i.e.*, if $\exists o'_1 \prec o'_2 \in \Omega(U_i), o''_2 \prec o''_1 \in \Omega(U_j)$, s.t. $o_1.(sw, key) = o'_1.(sw, key) = o''_1.(sw, key), o_2.(sw, key) = o'_2.(sw, key) = o''_2.(sw, key)$ and $i < j$, then remove element $o_1 \prec o_2$;
- Step 4.2: After removing $o_1 \prec o_2$, inherit the implicit dependencies between o'_1's parents and o'_2's children from the previous update U_i, *i.e.*, $\forall m, n$, if $\exists o_m \prec o'_1 \in \Omega(U_i)$ and $o'_2 \prec o_n \in \Omega(U_i)$, add elements $o_m \prec o_n$ into $\Omega(U_k \circ ... \circ U_1)$;
- Step 5: Simplify $\Omega(U_k \circ ... \circ U_1)$ to a minimal partial order set by removing redundant elements based on the transitivity of $\prec$.

Theorem 4. *If the sequential execution from $\Omega(U_1)$, $\Omega(U_2)$,..., to $\Omega(U_k)$ guarantees the basic consistency, the non-blacking execution of $\Omega(U_k \circ ... \circ U_1)$ in update algebra can guarantee the basic consistency.*

The detailed proof of Theorem 4 is omitted due to s-pace limitation. The intuition here is that the first two steps allow $\Omega(U_k \circ ... \circ U_1)$ to inherit the orders from $\{\Omega(U_1), \Omega(U_2), ..., \Omega(U_k)\}$. In Step 3, any cycle indicates the presence of order conflicts between U_i and U_j for the same flow rules. Since based on **Selectivity** in Theorem 1, the conflicting operations are overwritten by the last one, applying the same rationale into the order constraint, only the order element in U_j, $(i < j)$, is preserved as depicted in Step 4.1. Because of the transitivity property, many of the order dependencies are hidden and implicit in the order constraint. But to break cycles, we remove some elements, which may lead to the loss of these implicit dependencies. For the example in Fig. 6, if we remove $o_{e1} \prec o_{a1'}$ from $\Omega(U_2)$, the implicit dependency $o_{e1} \prec o_{b1'}$ will be lost. Therefore in Step 4.2, to avoid such loss, for each element we remove from $\Omega(U_k \circ ... \circ U_1)$, we inherit its implicit dependencies from the previous corresponding update. The goal of Step 5 is to remove redundant order elements in the constraint, *e.g.*, if Ω contains $o_1 \prec o_2$, $o_2 \prec o_3$ and $o_1 \prec o_3$, the last component $o_1 \prec o_3$ is redundant and should be removed.

Example. Fig. 6 shows the efficient composition of U_1 and U_2 in the example of Fig. 2. We use the format of DAGs for simple illustration. After Steps 1 and 2, there is a cycle between $o_{a1''}$ and $o_{b1'}$, where $o_{a1''} = o_{a1'} \circ o_{a1}$ is computed according to TABLE II. Since $o_{b1'} \prec o_{a1''}$ is inherited from $\Omega(U_1)$ but conflicts with $\Omega(U_2)$, it will be removed as depicted in Step 4.1 of our solution. This example does not involve Steps 4.2 and 5, but more interesting and complicated examples can be found in our technical report [17].

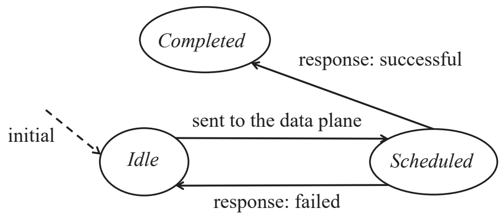


Fig. 7: Finite machine of an operation.

D. Composition with Partial Execution

The update algebra in the previous sections assumes that all operations of updates to be composed are not executed. However in practice, new updates can arrive while previous ones are partially executed. The problem is that such a partial execution results in uncertainty of configuration states. To resolve this problem, we 1) introduce an uncertainty model to reflect partial executions, and 2) provide a solution for continuous and non-blocking composition for updates with partial executions.

1) Uncertainty Model: An SDN controller sends operations of an update to a data plane (in a proper order) to execute the update. Once switches receive the operations, they reply to the controller with the progress of execution. Therefore, from the perspective of the SDN controller, each operation has one of the following states: *Idle*, *Scheduled* or *Completed*. Fig. 7 shows the finite machine of an operation state. Initialized with state *Idle*, once an operation is sent to the data plane for installation, its state becomes *Scheduled*. After installation, switches return a response message to notify whether the operation is applied successfully. A successful response turns the operation state into *Completed* state whereas a failed response turns it back to *Idle*.

The configurations according to the states of an operation $o : C \to C'$ are as follows:

- **Invariant 1:** If o is at *Idle*, it is not applied at the data plane; *i.e.*, the configuration is C.
- **Invariant 2:** If o is at *Completed*, it is applied at the data plane; *i.e.*, the configuration is C'.
- **Uncertainty:** If o is at *Scheduled*, it can be applied or not; *i.e.*, the configuration is C or C'.

Partial Execution. Consider an update composition of U_1 and U_2 in which U_1 is partially-executed when U_2 arrives at the controller. Let U_1^+ denote the part of U_1 that has been applied (*Completed*), and U_1^- the remainder, i.e., $U_1 = U_1^- \circ U_1^+$. Based on **Associativity** in Theorem 3, we have:

$$U_2 \circ U_1 = U_2 \circ (U_1^- \circ U_1^+) = (U_2 \circ U_1^-) \circ U_1^+ \qquad (8)$$

Fig. 8(a) presents the transitions of configuration states according to executed updates. Note that we compose U_1 and U_2 at the configuration state C'_1 with a partial execution U_1^+, which means $U_2 \circ U_1^-$ is our target composition. The problem is that C'_1 and U_1^- may be unknown to the controller due to the uncertainty state of operations at *Scheduled* state. A solution is

$$\begin{tikzcd}
C_0 \arrow[r, "U_1^+"] \arrow[dr, "U_1"'] & C_1' \arrow[d, "U_1^-"] \arrow[dr, "U_2 \circ U_1^-"] & \\
 & C_1 \arrow[r, "U_2"'] & C_2
\end{tikzcd}$$

(a) Uncertainty model.

$$\begin{tikzcd}
 & C_1^F \arrow[dd, "o_1"] \arrow[ddr, "U^S \circ o_1"] & \\
C_1' \arrow[ur, dashed, "o_1 \text{ failed}"] \arrow[dr, dashed, "o_1 \text{ succeeded}"'] & & \\
 & C_1^S \arrow[r, "U^S"'] & C_2
\end{tikzcd}$$

(b) Illustration of Problem 1.

Fig. 8: (a) Uncertainty model where C_1' and U_1^- may be unknown. (b) Illustration of Problem 1 where o_1 is at the *Scheduled* state, so its execution can be either failed or successful.

to prohibit *Scheduled* states during composition; *i.e.*, once U_2 arrives, the controller stops scheduling new operations in U_1 and collects responses for all *Scheduled* ones until their states become steady one such as *Idle* and *Completed*. However, this solution inefficiently blocks updates due to the interruption for collecting the responses.

2) Solution: The basic idea behind our solution is to treat *Scheduled* operations as "not applied" at a data plane during composition. Consider an update composition $U_2 \circ U_1$ in Fig. 8(a). Assume that one operation o_1 in the update U_1 is in *Scheduled* state. Suppose that C_1^F and C_1^S are the configurations after failed and successful responses for o_1, respectively. Based on the two **Invariants**, the current configuration C_1' must be either C_1^F or C_1^S. Then, we aim to solve:

Problem 1. *Given $C_1' \in \{C_1^F, C_1^S\}$, find $U : C_1' \to C_2$.*

Let $U^S : C_1^S \to C_2$ denote the composed update to achieve U_2. As depicted in Fig. 8(b), we have

$$C_1^S = o_1(C_1^F), \tag{9}$$

$$C_2 = U^S \circ o_1(C_1^F). \tag{10}$$

Recall that our solution treats *Scheduled* operations as "not applied", *i.e.*, $C_1' = C_1^F$. Based on the assumption $C_1' = C_1^F$, the solution is $U = U^S \circ o_1$ from Equation (10). We show that the solution yields $U(C_1') = C_2$ even in the case of $C_1' = C_1^S$ as follows:

$$\begin{aligned}
U^S \circ o_1(C_1') &= U^S \circ o_1(C_1^S) && (11)\\
&= U^S \circ o_1(o_1(C_1^F)) && (12)\\
&= (U^S \circ o_1) \circ o_1(C_1^F) && (13)\\
&= U^S \circ (o_1 \circ o_1)(C_1^F) && (14)\\
&= U^S \circ o_1(C_1^F) && (15)\\
&= C_2, && (16)
\end{aligned}$$

where Equation (12), (14), (15) and (16) are deduced from (9), **Associativity**, **Idempotent** in Theorem 1 and Equation (10), respectively.

Therefore, regardless of the uncertainty due to *Scheduled* operations, the update algebra is able to compute an update composition and achieve correctness based on our solution.

IV. Evaluation

This section evaluates the benefits of update algebra through asymptotic analysis (Section IV-A), extensive benchmarking using a real controller (Section IV-B), and integration with a real application (Section IV-C).

A. Asymptotic Analysis

We conduct an asymptotic analysis to compare the correctness, overhead, and completion time of the composition of consecutive updates using sequential, parallel, and update algebra based executions. We denote $p \in [0, 1]$ the probability that two consecutive updates are related, *i.e.*, involve common flows. In other words, $1 - p$ denotes the probability that two consecutive updates are fully independent. Each update is represented as a sequence of operations. For simplicity, if two consecutive updates U_1 and U_2 are related (with probability p), the two sequences are modeled as overlapping, and the common segment is randomly selected using a uniform distribution. The total length of the update after composition is provided by Equation (17), with N_1 and N_2 representing the lengths of U_1 and U_2. The details of the proof are omitted due to space limitation.

$$f(N_1, N_2) = \frac{N_1^2 + N_1 N_2 + N_2^2}{N_1 + N_2} - 1 \tag{17}$$

TABLE III summarizes the results. First, as explained in Section II, parallel execution may violate correctness as it does not respect update dependencies. Second, TABLE III shows that the completion time of a sequential execution increases linearly with the number of updates. In contrast, in independent-update dominant settings, the completion time with update algebra stays asymptotically constant, similar to that of a parallel execution, while guaranteeing correctness.

TABLE III: Asymptotic analysis of the composition of two consecutive updates U_1 and U_2.

	Correctness	Operation Number (N)	Update Completion Time (T)
Sequential	$\surd$	$N_1 + N_2$	$T_1 + T_2$
Parallel	$\times$	$N_1 + N_2$	$max(T_1, T_2)$
Update Algebra	$\surd$	$\approx f(N_1, N_2)p + (N_1 + N_2)(1 - p)$	$\approx f(T_1, T_2)p + max(T_1, T_2)(1 - p)$

B. Benchmarking

Methodology: We deploy update algebra in a SDN network running a Ryu 4.24 controller, and twenty Open vSwitch [18] 2.5.4, connected in a FatTree topology, which is common in data centers.

We generate updates as follows: each update involves a random number of flows ranging from 17 to 20. Two successive updates include common flows with a probability of p. We vary p, and for each case, we synthesize 300 network update events with different Poisson arrival rates λ.

The controller schedules the arriving updates according to the order constraints (*i.e.*, DAG) derived using the algorithm [9] proposed by Forster *et al.* to ensure a lack of blackholes and loops during the updates. We compare two composition approaches: in the first, the controller keeps track of the state of each operation, and continuously merges new arriving updates with ongoing ones through update algebra; in the second approach, the controller executes the arriving updates in a sequential (blocking) manner, *i.e.*, according to a First-In-First-Out (FIFO) policy.

To compare the performance, we report two metrics: *average operation number*, and *average update completion time*.

The former is the average number of operations applied in the data plane for the 300 updates. The latter is the average duration of an update which begins when it is considered by the controller, and ends when the last operation is completed or merged with new updates.

Results:

• *Control Overhead:* Fig. 9(a) shows the average number of operations, which reflects the control message overhead in the system. The sequential execution yields a constant number of operations for different values of λ as all updates are executed individually and sequentially, independent of their arrival rate. In contrast, the overhead of update algebra based execution varies with λ: the number of operations is similar to that of the sequential execution when λ is low, as each update completes before the next one arrives. However, as λ increases, fewer updates are completed before new ones arrive. Therefore, consecutive updates can be merged, reducing the number of operations. Comparing the results of different p, we observe that when successive updates are more related (*i.e.*, larger p), more operations can be merged through update algebra, resulting in a reduction in numbers of operations. As such, with $\lambda = 2$/s (the network being updated twice a second on average), the non-blocking execution in update algebra reduces the control message overhead by up to 30%.

• *Completion time:* Fig. 9(b) shows the average update completion time. It includes both the results from the experimental evaluation, and the theoretical upper and lower bounds.

For the theoretical bounds, we model the sequential execution as an M/M/1 system where the arrival rate is λ and the service rate μ is 1/(average update execution time); note that the execution time of individual updates is exponentially distributed in our experiments. For the parallel execution, as updates can be executed concurrently, the execution bottleneck comes from the operation queues at the switches. Considering that updates arrive at switches uniformly at random, the parallel execution can be approximated by an M/M/c system, where c is the number of switches in the network. Therefore, the average response time in the M/M/1 system reflects the upper bound of the update completion time, and that the M/M/c system corresponds to the lower bound.

Fig. 9(b) depicts the theoretical bounds, and the measured update completion times of both the sequential execution and update algebra. Both the sequential and update algebra based executions yield similar constant times at low arrival rates as all updates are completed without blocking. However, as λ becomes larger, our approach completes updates faster than the sequential execution. This is because the sequential execution suffers from long waiting times due to the blocking in the queue, while update algebra enables updates to be executed concurrently, and reduces the number of operations, *e.g.*, by merging operations with same keys. When $p = 0.2$ and $\lambda = 1.2$/s, update algebra improves network update speed by up to 8x compared to the sequential execution, and when $p = 0.8$ and $\lambda = 1.6$/s, the gain can reach 16x.

The close-up figure in Fig. 9(b) (right-hand) clearly shows the impact of different update patterns on the update performance. As p increases, update algebra achieves higher gain, and the update completion time with update algebra gets closer to that of the theoretical lower bound. The results demonstrate that by maximizing the execution parallelism while preserving consistency properties, update algebra can handle frequent network changes in a non-blocking, but also efficient way.

C. *Performance in a Real Application*

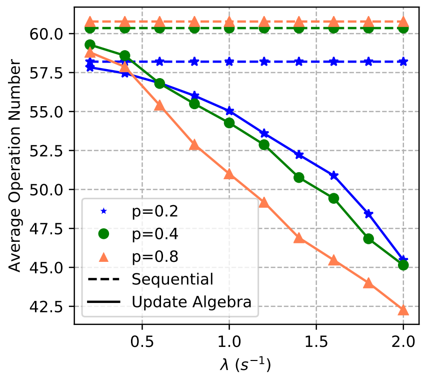
This section evaluates the performance benefits of update algebra integrated with a real application. We deploy Hedera [2] in the SDN controller. Hedera continuously collects network statistics, and updates flow routes to maximize the aggregate network utilization. We use an unbalanced traffic pattern with a large workload, and measure the average link bandwidth utilization with different update frequencies. The update frequency is an adjustable parameter (with a default value 0.2/s) of Hedera specifying how frequently the updates are generated.

Fig. 10 shows that the update algebra based execution outperforms the sequential execution. As the update frequency increases, more updates are generated. The sequential execution cannot complete the updates before the next ones arrive. As a result, the arriving updates accumulate and the controller fails to update the network as directed by Hedera, causing network performance to degrade quickly. This is the reason frequent updates are prohibited in current systems, and the default frequency is set to 0.2/s (an update every five seconds). In contrast, by merging consecutive updates in a non-blocking manner, update algebra allows updates to be quickly enforced, and the network performance keeps increasing with the update frequency. With an update frequency of 1.2/s, update algebra increases the network utilization by 13% compared to the default value 0.2/s, and outperforms the sequential execution by 30%. These results demonstrate the benefits of update algebra with a real application.

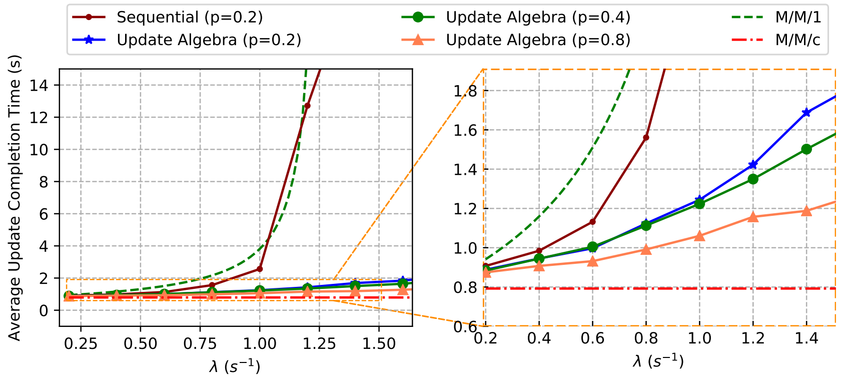
V. Related Work

Consistent Updates: A concerted research effort has recently been made to tackle the problem of network updates in SDN for different aims. Xitao *et al.* [8] minimize the number and latency of rule updates for TCAM-based switches by eliminating redundant and unnecessary entry moves. Reitblatt *et al.* [19] introduce the notion of consistent network updates, and propose a two-phase update approach. Solutions supporting a broad range of consistent properties are proposed, including loop freedom [9], congestion-freedom [10], waypoint routing [11] and customizable properties [12]. However, existing work is limited to a single network update at a time, and does not handle consecutive network updates. Peter *et al.* [13] mention the inter-update scheduling problem in their future work, but only provide a strawman solution as an enhancement to the sequential approach. In contrast, our work allows controllers to efficiently merge multiple network updates to handle continuous and non-blocking network changes while preserving desired properties. To the best of the authors' knowledge, our work is the first to handle multiple updates as a group.

Policy Composition: Researchers have also investigated composing policies. Several recent SDN policy languages and controller hypervisors (*e.g.*, NetKAT [20], Pyretic [21]) support taking multiple high-level policies and generating flow tables that fulfill the semantics of the sequential and parallel composition. However, network update operations are different from flow rules, and present unique challenges as well as distinct requirements. For example, as discussed in



(a) Average operation number.



(b) Average update completion time. The right figure is a close-up view of a specific area from the left figure.

Fig. 9: Benchmarking results.

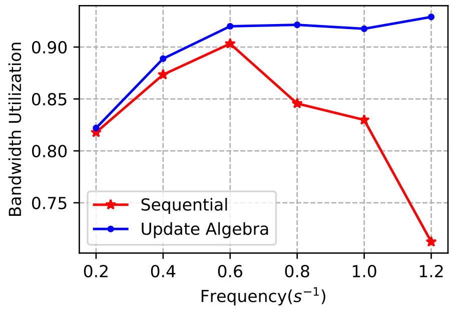


Fig. 10: Bandwidth utilization of Hedera with sequential execution and update algebra.

Section III-D, network updates may be partially executed, and controllers may not have a complete and precise up-to-date view of the update progress. In addition, network updates have distinct consistency requirements that differ from those of policy composition. Consequently, existing work on policy composition cannot be applied to composing network updates. Instead, we developed a theoretical framework that captured the unique characteristics of network updates, and allowed us to reason about their properties and composition.

Acknowledgment

We would like to thank Sanat Khurana who found a counter example of our earlier algorithm and helped with the design of a new solution. This research was supported in part by NSFC #61701347, NSFC #61702373 and NSFC #61672385; NSF grant #1440745, CC*IIE Integration: Dynamically Optimizing Research Data Workflow with a Software Defined Science Network; Google Research Award, SDN Programming Using Just Minimal Abstractions. This research was also sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001.

References

[1] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven Networking: A Deep Reinforcement Learning based Approach," *INFOCOM*, 2018.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks." in *Proc. of NSDI*, 2010.

[3] Q. Xiang, H. Yu, J. Aspnes, F. Le, L. Kong, and Y. R. Yang, "Optimizing in the dark: Learning an optimal solution through a simple request interface," in *AAAI*, 2019.

[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. of SIGCOMM*, 2014.

[5] G. Li, Y. Qian, L. Liu, and Y. R. Yang, "JMS: Joint Bandwidth Allocation and Flow Assignment for Transfers with Multiple Sources," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. IEEE, 2018, pp. 123–130.

[6] Q. Xiang, J. J. Zhang, X. T. Wang, Y. J. Liu, C. Guok, F. Le, J. MacAuley, H. Newman, and Y. R. Yang, "Fine-grained, multi-domain network resource abstraction as a fundamental primitive to enable high-performance, collaborative data sciences," in *Proceedings of SC*, 2018.

[7] S. Sinha, S. Kandula, and D. Katabi, "Harnessing TCPs Burstiness using Flowlet Switching," in *Proc. of HotNets*, San Diego, CA, 2004.

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

[9] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes," in *Proc. of IFIP Networking*, 2016.

[10] W. Wang, W. He, J. Su, and Y. Chen, "Cupid: Congestion-free consistent data plane update in software defined networks," in *INFOCOM*, 2016.

[11] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proc. of HotNets*, 2014.

[12] W. Zhou, D. K. Jin, J. Croft, M. Caesar, and P. B. Godfrey, "Enforcing customizable consistency properties in Software-Defined Networks." in *Proc. of NSDI*, 2015.

[13] P. Pereíni, M. Kuzniar, M. Canini, and D. Kostić, "ESPRES: transparent SDN update scheduling," in *Proc. of HotSDN*, 2014.

[14] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[15] D. S. Dummit and R. M. Foote, *Abstract algebra*. Wiley Hoboken, 2004.

[16] G. Li, Y. Qian, C. Zhao, Y. R. Yang, and T. Yang, "DDP: Distributed Network Updates in SDN," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018.

[17] G. Li, Y. R. Yang, F. Le, Y. sup Lim, J. Wang, and S. Khurana, "Update Algebra: Toward Continuous, Non-Blocking Composition of Network Updates in SDN (extended version)," https://cpsc.yale.edu/sites/default/files/files/tr1548.pdf, Yale Technical Report TR1548, Tech. Rep.

[18] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The Design and Implementation of Open vSwitch." in *Proc. of NSDI*, 2015.

[19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," *Proc. of SIGCOMM*, 2012.

[20] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *ACM SIGPLAN Notices*, 2014.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing Software Defined Networks." in *Proc. of NSDI*, 2013.