Enforcing Generalized Consistency Properties in Software-Defined Networks

Wenxuan Zhou[*], Dong Jin[**], Jason Croft[*], Matthew Caesar[*], and P. Brighten Godfrey[*]

[*]University of Illinois at Urbana-Champaign
[**]Illinois Institute of Technology

Abstract

It is critical to ensure that network policy remains consistent during state transitions. We propose the Generalized Consistency Constructor (GCC), a fast and generic framework to support *customizable* consistency policies during network transitions under temporal uncertainty about the network state. GCC effectively reduces the task of synthesizing an update plan under the constraint of any given consistency policy to a general verification problem, by checking whether an update can safely be installed in the network at a particular time and greedily processing network state transitions to heuristically minimize transition delay. We identify the scope of consistency policies that are guaranteed by GCC's heuristic. In addition, GCC's greedy update scheduling algorithm can easily be integrated with existing network update mechanisms for significant speed improvements with the same level of consistency enforcement. In that way, GCC nearly achieves the "best of both worlds": the efficiency of simply passing through updates in most cases, with the consistency guarantees of more heavyweight techniques. Both Mininet and physical testbed evaluations demonstrate GCC's capability to achieve various types of consistency, such as path and bandwidth properties, with zero switch memory overhead and up to a factor 3x delay reduction compared to previous solutions.

1 Introduction

Network operators often establish a set of correctness conditions to ensure successful operations of the networks. Some of those are manually expressed in the form of policies, i.e., configurable rules, or regulations on how the network should behave, such as the preference of one path over another, or the prevention of untrusted traffic from entering a secure zone. Others are deemed so obvious they are "baked in" to protocols that run in networks, such as the need to avoid forwarding loops or black holes. As networks become an increasingly crucial backbone for critical services, such as the power grid, medical networks, military and rescue operations, and financial services, and as network operators face increasing regulatory requirements on correctness and security properties of data transport (e.g., data isolation as in HIPAA, Sarbanes-Oxley, etc.), the ability to construct networks that obey correctness criteria is becoming even more important.

Unfortunately, today's approach of instilling policies on networks is an error-prone process. Operators need to program policies into networks via indirect mechanisms (low-level configuration commands on routers), and must coordinate actions of complex protocols running across large numbers of distributed devices to achieve their network-wide objectives. The challenge is to preserve a specific property for all network traffic even during network state transitions, which we refer to as maintaining *network consistency*. Researchers have explored how to consistently update a network to new states [15, 19, 23], which could potentially move a network between two operator-specified network snapshots. However, those methods do not support maintenance of general operator-supplied policies during the transition process, and/or substantially delaying the network update process.

That leads to a question: is it possible to efficiently maitain *customizable* correctness policies as the network evolves? To achieve that goal in the context of Software-Defined Networks (SDN), we propose a generic framework, Generalized Consistency Constructor (GCC), which is centered around the idea of converting complex scheduling problems to well-defined network verification problems.

Operators can express customized network properties, and GCC ensures that those properties are efficiently maintained in the presence of network dynamics. GCC works by constructing a model of distributed network operations, and automatically synthesizing update sequences to enforce a set of supplied *network invariants* by verifying the model against the invariants.

Dionysus is perhaps the closest work to ours, as it attempts to reduce network update time to just what is necessary to satisfy a certain property. However Dionysus requires a dependency graph for each particular invariant, produced by an algorithm specific to that invariant (the paper presents an algorithm for packet coherence). For example, a waypoint invariant would need a new algorithm. In GCC, the network programmer needs only to specify how to *check* the property, rather than the algorithmic design task of producing a dependency graph automatically. Further illustrating that point, note that the algorithms of [**?**] work only when forwarding rules match exactly one flow; the more complicated case when rules overlap, such as with longest prefix match, becomes trivial in GCC since checking properties is easy.

Because of the distributed nature of networks, the GCC engine is unaware of the precise ordering and timing with which updates arrive at devices in the network. To address that, GCC explicitly models the "uncertainty" about network states through the use of a novel *symbolic network representation*, a data structure that compactly represents all different possible network states. To maximize update speed, we have developed a greedy heuristic scheduling algorithm in GCC, and precisely identify the scope of policies whose consistency is completely guaranteed by the heuristic. To make GCC capable to handle any given policies, we designed a fallback algorithm that allows GCC to fall back from the greedy heuristic to more heavyweight techniques, such as Consistent Updates [23] and SWAN [12]. GCC guarantees such fallback is triggered only when theoretically required, thus preserving strong and flexible consistency properties, with significant reductions in time and memory in practice. Through both emulations and physical testbed experiments, we have demonstrated GCC can achieve ...

2 Problem Definition and Related Work

To ensure that networks are always in the correct states over time and network state changes, we design GCC to achieve the following three objectives.

1) Consistency at Every Step. The asynchronous and distributed nature of modern networks implies that no single network component can always obtain a correct instantaneous network-wide view. The emergence of SDN, despite providing a logically centralized management interface, does not change that fact [6]. After issuing updates to the network, the controller has limited knowledge about the exact timing and ordering with which the updates will be applied. After network changes occur, such as device failures, link congestion, and end-hosts migrations, the controller will be aware of those changes only after a certain delay, or may never learn of them, depending on the implementation. Moreover, data packets from all possible sources may traverse the network at any time in any order, interleaving with the network data plane updates. How can we enforce consistency properties at every step of state changes, given the incomplete and uncertain network view at the controller? A class of work [12, 15, 19, 22, 23] carefully transitions network states in such a way that any intermediate state complies with the required properties. However, those solutions do not handle generalized properties, and/or produce slow updates.

2) Efficient Update Installation. Network changes in SDN occur frequently, triggered by the control applications, changes in traffic load, system upgrades, or even failures. Accordingly, they require timely reactions to minimize the duration of performance drops and network errors. There have been proposals [12,15,19,21,23] that instill correctness according to a specific consistency property, but suffer substantial performance penalties in practice. For example, the total waiting time of the two-phase update scheme proposed in CU [23] is at least the maximum delay across all the devices, assuming a completely parallel implementation. Dionysus [14] is recently proposed to efficiently update networks via dynamic scheduling on top of a consistency-preserving dependency graph. For each particular invariant, Dionysus requires a dependency graph and an algorithm specific to that invariant. For example, a packet coherence invariant needs one algorithm and a waypoint invariant would need a new algorithm. Our approach is to convert the complex scheduling problems to general network verification problems, in which operators only need to specify the verification function instead of designing a new algorithm to automatically generate the dependency graph. For example, it is straightforward to handle overlapped rules (with longest prefix match) in GCC than in Dionysus, since verifying the properties is easy.

3) Generalized Consistency Properties. The range of consistency properties of networks is very broad, for instance, the successful operations of some networks depend on loop freedom, balanced load across links, use of optimal routes, enforcement of access control to secure critical assets, or a combination of several of these. As argued in [20], a generic framework to handle general properties is needed. Researchers have attempted to ensure certain types of consistency properties, e.g., loop freedom, packet coherence, absence of packet loss [12, 15, 19, 19, 23], but those studies do not provide a generalized solution. Dionysus [14], as stated earlier, generalizes the scope of consistency properties it deals with, but still requires designing specific algorithms for different invariants. There are also tools [4, 16, 18] that attempt to check network state snapshots incrementally for *general* properties, and optionally block faulty updates, but they do not instill correctness. The difference between our work and that earlier work is that we focus on the *generality* of the consistent properties that a system needs to maintain as well as the high *efficiency* (fast update speed and small memory requirement) needed to process network updates.

3 Generalized Consistency Constructor (GCC)

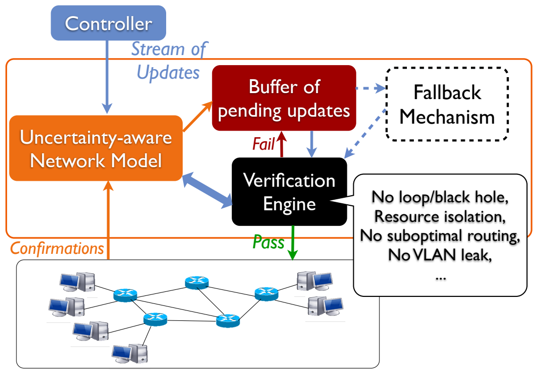

Controller
Stream of Updates
Buffer of pending updates
Fallback Mechanism
Uncertainty-aware Network Model
Fail
Verification Engine
No loop/black hole, Resource isolation, No suboptimal routing, No VLAN leak, ...
Confirmations
Pass

Figure 1: ***System architecture of GCC.***

To efficiently enforce consistency of network-wide invariants and policies as network states evolve, we covert the complex update scheduling problem into a well-defined network verification problem. However, verifying every possible update ordering would be very inefficient. What's more, at any instance of time, due to the inherent "network uncertainty" (defined in §4), one has to check all possible views of the network to eliminate any possible policy violations.

To deal with the simultaneous possible network views, we cannot utilize existing network verifiers, such as VeriFlow [18] or HSA [17], which effectively yet inaccurately assume that the updates are applied at the exact moment when the verification tools see them. In other words, they only check static network snapshots, and do not consider temporal uncertainty during the transition of the snapshots. In contrast, GCC explicitly models the *uncertainty* about network state with a novel symbolic network representation — a data structure that compactly represents all different possible network states.

Yet, another challenge remains. How can we efficiently find a feasible update sequence that preserves the desired policies among all possible sequences?GCC uses a greedy heuristic algorithm as the default scheduling algorithm. The algorithm utilizes the network verifier to determine whether an update is safe to pass to the network at a given time; it then greedily processes the network state transition, and heuristically maximizes the parallelism of updating the network. We formally identify the large scope of consistency policies that are guaranteed by GCC's heuristic (§5.3). GCC achieves high update efficiency (in both speed and memory consumption) by starting handling updates with the heuristic, and falling back to other heavyweight techniques (e.g., Consistent Updates [23] and SWAN [12]) only when necessary (very rare in practice as shown in §7), to guarantee any given consistency policies. Figure 1 depicts the system architecture. One key feature of GCC is that as a platform, it operates in a black-box fashion. Therefore, it is flexible for network operators to "plugin" a different verification engine and an update scheduling tool to meet different needs.

4 Uncertainty-aware Network Modeling

We start by describing the problem of network uncertainty (§4.1), and then present our solution to *model* a network in the presence of uncertainty (§4.2 and §4.3). Our design centers around the idea of *symbolic graphs*, which compactly represent the entire set of possible network states from the standpoint of packets in flight.

4.1 Network Uncertainty

We first describe the problem of network uncertainty and its negative effect on network-wide verification. It takes time in networks to disseminate states among distributed and asynchronous devices, which leads to the inherent *uncertainty* that an observation point knows the current state of the network. We refer to the time period during which the view of the network from an observation point (e.g., an SDN controller) might be inconsistent with the actual network state as *temporal network uncertainty*. The uncertainty could cause network behaviors to deviate away from the desired invariants temporarily or even permanently.

Figure 2 shows a motivating example. Initially, switch *A* has a forwarding rule directing traffic to switch *B*. Now the operator wants to reverse the flow of traffic by issuing two instructions in sequence: (1) remove the forwarding rule in *A*, and (2) insert a new forwarding rule (directing traffic to *A*) in *B*. Because of the network uncertainty, it is possible that the second operation will finish earlier than the first one, causing a short-term loop that leads to increased traffic load as well as packet losses. That is not an uncommon situation; for example, three out of eleven bugs found by NICE [6] (BUG V, IX and XI) are caused by the control programs' lack of knowledge of the network states.

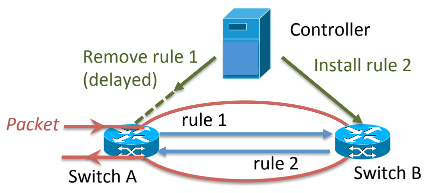


Figure 2: ***Example: challenge of modeling networks in the presence of uncertainty.***

Such errors may have serious consequences. For example, as in the previous example, a packet may enter switch *A* while the forwarding loop exists. As switches typically drop packets instead of forwarding them back through their ingress ports, the packet will encounter a black hole at switch *B* instead of a loop. That could result in a significant performance drop. E.g., a recent study [8] shows that TCP transfers with loss may take five times longer to complete. Such errors could also violate security policy, e.g., malicious or untrustworthy packets could enter a secure zone because of a temporary access control violation [23].

To make matters worse, errors caused by network temporal uncertainty can be permanent . For instance, a control program initially instructs a switch to install one rule, and then instructs the switch to remove that rule. The problem is that the two instructions can be reordered at the switch [10], which ultimately leads the switch to install a rule that ought to be removed. The view of the controller and the network state will remain inconsistent until the rule expires. One may argue that inserting a barrier message in between the two instructions would solve the problem. However, serializing rule orderings with barrier messages may greatly harm performance because of increasing control traffic and switch operations. There are also scenarios in which carefully crafting ordering does not help [23]. In addition, it is difficult for a controller to figure out when to insert the barrier messages. GCC addresses that by serializing only updates that have potential to cause race conditions (§**??**).

4.2 Uncertainty Model

To address the problem, we construct a model that accurately represents the controller's uncertainty about the network state, along with a checking process to allow the controller to make decisions on when and how to send updates by checking that model.

Naively, to model the network uncertainty, for every update, we need two graphs to symbolically represent the network behaviors with and without the effect of the update, until the controller is certain about the status of the update. With that approach, if n updates were concurrently "in flight" from the controller to the network, it would take 2^n graphs to represent all possible sequences of update arrivals. Such a state-space explosion will result in a huge memory requirement and excessive processing time to determine consistent update orderings.

To address that issue, our approach efficiently models the network forwarding behavior as an *uncertain* graph, in which links can be marked as *certain* or *uncertain*. The effect of a forwarding link is marked as *uncertain* if the controller does not yet have information on whether that corresponding update has been applied to the network. The graph is maintained by the controller over time. When an update is sent, its effect is applied to the graph and marked as uncertain. After receipt of an acknowledgment from the network that an update has been applied (or after a suitable timeout), the state of the link can be modified to become certain. By inspecting the graph, the controller becomes aware of which states in the network are reliably known, whether a newly arrived update from an application could result in an inconsistency with in-flight updates, and so on.

One complication is that routers maintain multiple forwarding rules, each of which can affect different subsets of packets. Hence, representing uncertainty on the granularity of physical links is not sufficient; an update may affect only one subset of packets traversing a link. Therefore, GCC maintains a *set* of uncertainty graphs. To minimize the number of graphs we need to store, we maintain only one graph per *equivalence class*, i.e., per each set of packets that undergo the same forwarding behavior throughout the network. Our notion of equivalence classes is similar in spirit to those used by HSA and VeriFlow, but applied to our uncertainty graph.

Note that it takes much less space to store the uncertainty graph than to store all possible network states explicitly, although it still takes more space than would be needed simply to store a static snapshot of the network. For example, an OpenFlow device typically handles an incoming packet with the highest-priority rule that the packet header matches. GCC collects not only the rule with the highest priority, but also the rules from the highest priority to lower priorities until a *certain* rule (included) is found, because any rule in this collection may be used to forward packets at the moment. The extra storage GCC requires because of uncertainty modeling is linearly bounded by the number of uncertain rules, and so is the query time. The reason is that in the worst case, there are n parallel paths that need to be traversed, where n is the number of concurrent uncertain rules. Such an uncertain graph is the representation of all the possible combinations of forwarding decisions at all the network devices. That leads to the question of when it is possible to "garbage collect" state from the graph, i.e., when one can "confirm" links as being certain (§ 4.3).

4.3 Dynamical Updating of the Model

In order to model the most up-to-date network state, we need to update the model as changes happen in the network. At first glance, one might think that could be done simply by marking links as uncertain when new updates are sent, and then, when an ack is received from the network, marking them as certain. The problem with that approach is that such a way may result in inconsistencies from the data packets' perspective, e.g., even after a switch removes a rule, packets that had been processed by that rule might remain in flight.

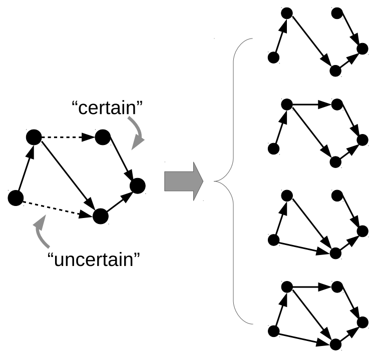


Figure 3: ***GCC's symbolic network representation provides a compact way to represent the uncertainty in a monitored network state.***

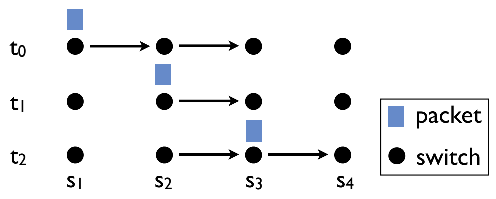


Figure 4: ***Example: Challenge of dealing with non-atomicity of network updates.***

Consider a network consisting of four switches, as in Figure 4. The policy to enforce is that packets from a particular source entering Switch s_1 should not reach Switch s_4. Initially, at time t_0, Switch s_3 has a filtering rule to drop packets from that source, whereas all the other switches simply pass packets through. The operator later wants to drop packets on s_1 instead of s_3. To perform the transition in a conservative way, the controller first adds a filtering rule on s_1 at t_1, and then removes the filtering rule on s_3 at t_2, after the first rule addition has been confirmed.

The forwarding graphs at all step seem correct. However, if a packet enters s_1 before t_1 and reaches s_3 after t_2, it will reach s_4, which violates the policy. The reason is that neither the installation of a set of updates nor the traversal of a packet is atomic. They interleave with each other. Note that we are not the first to observe that problem [23]. To deal with it, upon receiving an ack from the network, GCC does not immediately mark the state of the corresponding forwarding link as certain. Instead, it delays application of the confirmation to its internal data structure. In fact, confirmations of additions of forwarding links in the graph model can be processed immediately, and only confirmations of removals of forwarding links need to be delayed. The reason is that even after a forwarding rule has been deleted, packets that had been processed by the rule may still exist in the network, buffered in an output queue of that device, in flight, or on other devices. Such confirmations include acknowledgments of deletion, expiration, or replacement by a higher-priority rule on the same device.

As for how long confirmations should be delayed, one way is to use a fixed timeout, assuming we have an estimation of the upper bound of packet- or flow-level traversal latency. After such a delay, all packets that may be handled using the old state are drained off the network. Another possible way is to actively query the network to see if the packet or flow leaves the network. More details of how the state of a rule can be changed in our model are presented in § **??**. Thus, our uncertainty-aware model is able to accurately capture the view of the network from the packets' perspective.[1]

5 Verification and Consistency under Uncertainty

In this section, we describe how to leverage our model to build useful tools. First (§5.1), we describe how our model can be used to perform safe, uncertainty-aware network verification. Then (§5.2 §5.4), we describe how to leverage our model to efficiently synthesize update sequences that obey a set of provided invariants.

5.1 Verification

If our model is used, construction of a *correct* network verification tool is relatively straightforward. In particular, by traversing the uncertainty graph model using directed graph algorithms, we can infer whether a reachable path exists between a pair of nodes. That can be done in a manner similar to that used by existing network verification tools like HSA and Veriflow. However, the traversal process needs to be modified to take into account uncertainty. That is, when the traversal process traverses an uncertain link, we need to keep track of the fact that downstream inferences lack certainty. For example, if the traversal reaches a node with no *certain* outgoing links, it is possible that packets will encounter a blackhole even with multiple *uncertain* outgoing links available. Thus, we are able to answer queries by traversing the graph once.

Like Veriflow, our tool can determine whether a supplied invariant is violated, and, if so, output a counterexample (e.g., the specific set of forwarding table entries that caused the problem). Unlike Veriflow, our tool can also reason correctly in the presence of uncertainty. If the analysis can be conducted with certainty, our tool outputs the result. If it cannot (e.g., if multiple possible network states exist), our tool can output the set of possible different answers.

[1]This theorem is presented in a technical report that has been mailed to the program committee chairs.

5.2 Enforcing Correctness with Maximized Parallelism

As the ultimate goal of our system is to *instill* flexible and user-specified notions of correctness during network transitions, GCC provides an algorithm for synthesizing update sequences to networks that greedily *maximizes parallelism* while simultaneously obeying flexible consistency properties (Algorithm 1).

Whenever an update u is issued from the controller, GCC intercepts it before it hits the network. Network forwarding behavior is modeled as an uncertainty graph ($G_{uncertain}$) as described previously. Next, the black-box verification engine takes the graph and the new update as input, and performs a computation to determine whether there is any possibility that the update will cause the graph state to violate any policy internally specified within this engine. If the verification is passed, the update u is sent to the network and also applied to the network model $Model_{uncertain}$, but marked as uncertain. Otherwise, the update is buffered temporarily.

When a confirmation of u from the network arrives, GCC also intercepts it. The status of u in $Model_{uncertain}$ is changed to certain, either immediately (if u doesn't remove any forwarding link from the graph), or after a delay (if it does, as described in §4.3). The status change of u may allow some pending updates that previously failed the verification to pass it. Each of the buffered updates is processed through the routine of processing a new update, as described above. There are several possible optimizations. If the desired property needs to be enforced only on flows that match the same pattern, then upon confirmation of a previously issued update, only those buffered updates that match the same pattern need to be reprocessed by the verification engine. Or, for problematic updates, the verification engine can report the dependency relationship between the new updates and in-flight updates. Then, confirming an update will trigger the verification engine to check only those buffered updates that depend on the confirmed update. However, those optimizations require that the verification engine reveals extra information. To make the design policy-agnostic, we stick with the pure black-box approach, and evaluation results show that it performs well in practice.

Algorithm 1 Maximizing network update parallelism

```
SCHEDULE_INDIVIDUAL_UPDATE($Model_{uncertain}$, $Buf_{pending}$, $u$)
  On issuing $u$:
  $G_{uncertain}$ = ExtractGraph($Model_{uncertain}$, $u$)
  $verify$ = BlackboxVerification($G_{uncertain}$, $u$)
  if $verify$ == PASS then
      Issue $u$
      Update($Model_{uncertain}$, $u$, $uncertain$)
  else
      Buffer $u$ in $Buf_{pending}$

  On confirming $u$:
  Update($Model_{uncertain}$, $u$, $certain$)
  $Issue\_updates \leftarrow \emptyset$
  for $u_p \in Buf_{pending}$ do
      $G_{uncertain}$ = ExtractGraph($Model_{uncertain}$, $u_p$)
      $verify$ = BlackboxVerification($G_{uncertain}$, $u_p$)
      if $verify$ == PASS then
          $Buf_{pending}$ removes $u_p$
          Update($Model_{uncertain}$, $u_p$, $uncertain$)
          $Issue\_updates \leftarrow Issue\_updates + u_p$
  Issue $Issue\_updates$
```

Algorithm 2 Synthesizing update orderings

```
SCHEDULE_UPDATES($Model_{uncertain}$, $Buf_{pending}$, $U$,
$FB$, $T_{threshold}$)
  for $u \in U$ do
      SCHEDULE_INDIVIDUAL_UPDATE($Model_{uncertain}$,
  $Buf_{pending}$, $u$)

  On timeout($T_{threshold}$):
  $\tilde{U}$ = Translate($Buf_{pending}$, $FB$)
  for $u \in \tilde{U}$ do
      SCHEDULE_INDIVIDUAL_UPDATE($Model_{uncertain}$,
  $Buf_{pending}$, $u$)
```

5.3 Segment Independence

In this subsection, we identify consistency policies that guarantee the existence of feasible update orders, and prove that GCC's heuristic is able to find one such order. As defined in [23], *trace properties*, which characterize the paths that packets traverse through the network, cover many common network properties, including basic reachability, access control, loop freedom, VLAN leak freedom, and waypointing, to name a few. Our discussion will first focus on trace properties, and then extend to other properties, such as congestion freedom. Thus, a network configuration can be expressed as a set of paths that packets are allowed to take. A forwarding graph is a collection of the paths, and a configuration transition is equivalent to a transition from an initial forwarding graph, G_i, to a final graph, G_f, through a series of transient graphs, G_ts. Starts from two specific trace properties: loop freedom and black-hole freedom.

Loop and black-hole freedom For loop freedom, it is proved in [9] (Theorem 1-3) that given that both G_i and G_f are loop-free, during the transition, it is safe to update a node in a G_t (causing no loop) if that node satisfies one of the following two conditions: (1) in G_t it is a leaf node, or all its upstream nodes have been updated with respect to G_f; or (2) in G_f it reaches the destination directly, or all its downstream nodes in G_f have been updated with respect to G_f. Furthermore, if there are several updatable nodes in a G_t, then any update order among these nodes is loop-free. More importantly, in any loop-free G_t that is not equal to G_f (including G_i), there is at least one node that is safe to update. That is, there always exists a loop-free update order.

Similarly, for the black-hole freedom property, we have the following three theorems.

Theorem 1. *(Updatable condition): A node update does not cause any transient black-hole, if in G_f, it reaches the destination directly, or in G_t, all its downstream nodes in G_f have already been updated.*

Theorem 2. *(Simultaneous updates): If there are several updatable nodes in a G_t, any update order among these nodes is black-hole-free.*

Theorem 3. *(Existence of a black-hole-free update order): In any black-hole-free G_t that is not G_f (including G_i), at least one of the nodes is updatable, i.e., there is a black-hole-free update order.*

Any update approved by GCC always results in a consistent transient graph, so there exists an update order from that transient graph to the final graph, to ensure loop and black-hole freedom.

Generalized Trace Properties To get a uniform abstraction for trace properties, let us first visit the basic connectivity problem: node A should reach node B ($A \rightarrow B$). To make sure there is a connectivity between two nodes, both black-hole and loop freedom are needed. Obviously, black-hole freedom is downstream-dependent (Theorem 1), whereas loop freedom is upstream- (updatable condition (1)) **or** downstream-dependent (updatable condition (2)), and thus weaker than black-hole freedom. In other words, connectivity is a downstream-dependent property, i.e., updating from downstream to upstream is sufficient to ensure it.

A number of trace properties can be divided into basic connectivity problems, such as:

- Waypointing: packets should reach a waypoint W;
- Isolation: packets flowing between isolated domains must encounter a filter/drop node;
- Middlebox chain: packets must traverse a chain of middleboxes, e.g. $A \rightarrow B \rightarrow C$

Combinations of properties must be considered as well. For example, ”packets from A must reach its destination B, and traverse a waypoint W before reaching B”, can be expressed as: $A \rightarrow W \rightarrow B$.

For properties with points on the path that must be traversed ($A \rightarrow B \rightarrow C$), we refer to these points as *waypoints*. Then n waypoints cut both the old and new paths into $(n-1)$ **segments**.

Definition 1. Waypoints-based trace property: *A property that enforces a group of packets to go through a set of waypoints (including the source and destination) in a particular order.*

If there is no dependency between segments (Figure 5 (a)), then each of them can be updated independently based on downstream dependency. That suggests that for paths with no inter-segment dependencies, a property-compliant update order always exists. However, whether or not there exist dependencies among segments is determined by both the policy and topology.

Definition 2. Dependencies between segments: *Suppose n waypoints cut the both before and after path into $(n-1)$ segments: $old_1, old_2, ..., old_{n-1}$ and $new_1, new_2, ..., new_{n-1}$. If new_j crosses old_i ($i \neq j$), then the update of segment j is* **dependent** *on the update of segment i, i.e., segment j cannot start to update until segment i's update has finished, in order to ensure the traversal of all waypoints.*

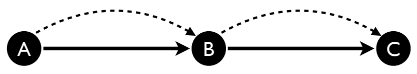
Otherwise, if segment j starts to update before i has finished, there might be violations. If $j < i$, there might be a moment when the path between waypoints j and $i+1$ consists only of new_j and part of old_i, i.e., waypoints $(j+1)...i$ are skipped. As in Figure 5(b), B may be skipped if the AB segment is updated before BC, and the path is temporarily $A \rightarrow 2 \rightarrow C$.

If $j > i$, there might be a moment when the path between waypoints $i, (j+1)$ consists of $old_i, old_{i+1}, ..., new_j$, and a loop is formed. As in Figure 5(c), the path could temporarily be $A \rightarrow B \rightarrow 1 \rightarrow B$.

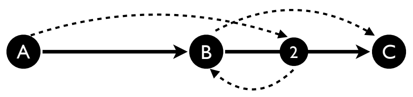
A special case is circular dependency between segments, as depicted in Figure 5(d), in which no feasible order exists.

Theorem 4. *(Condition of the existence of an invariant compliant update order): There are no circular dependencies between segments. In particular, if invariants are enforcing no more than two waypoints, an update order always exists.*

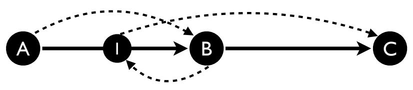
Figure 5(a) shows us one example forwarding graph that satisfies the above condition. However, in reality, forwarding links and paths may be shared by different sets of packets, e.g., multiple flows. Thus it is possible that two forwarding links (smallest possible segments) l_1 and l_2 will have conflicting dependencies when serving different groups of packets, e.g., in forwarding graphs destined to two different IP prefixes. In such cases, circular dependencies are formed across forwarding graphs. Fortunately, there are many cases, in which forwarding graphs do not share links. For example, as pointed out in [14], for many flow-based traffic management applications for the network core (e.g., ElasticTree, MicroTE,



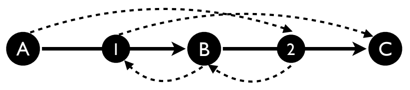
(a) Case 1: no crossing, update different segments in parallel, as long as each segment's updating follows downstream dependency



(b) Case 2: new *AB* crosses old *BC*, so *AB* depends on *BC*



(c) Case 3: new *BC* crosses old *AB*, so *BC* depends on *AB*



(d) Case 4: new *BC* crosses old *AB*, and new *AB* crosses old *BC*, so *BC* and *AB* have circular dependency between themselves.

Figure 5: ***Examples: dependencies between segments. Path AC is divided into two segments AB and BC by three waypoints A, B, and C, with old paths in solid lines, and new paths in dashed lines.***

B4, SWAN [5, 11–13]), any forwarding rule at a switch matches at most one flow.

Other Properties There are trace properties which are not waypoint-based, such as, quantitative properties like path length constraint. In order to preserve such properties and waypoint-based trace properties that are not segment independent, GCC is plugged in with Consistent Updates [23] as backup to its heuristic component. Also, there are network properties beyond trace properties, such as quality of service, and congestion freedom. Among those, congestion freedom is a common and important one. However, it has been proven that careful ordering of updates cannot always guarantee congestion freedom [12, 25]. Thus, in GCC, to ensure congestion freedom, we plug in a heavyweight tool (e.g., SWAN, which is the one used in our implementation) to work together with the heuristic algorithm.

5.4 Synthesis

As discussed previously, there may be situations in which this greedy algorithm gets stuck, i.e., some buffered updates never pass the verification. That may happen when desired policies do not have the segment-independence property (§5.3). For instance, consider a circular network with three nodes, in which each node has two types of rules: one type to forward packets to destinations directly connected to itself, and one default rule, which covers destinations connected to the other two switches. Initially, default rules point clockwise. They later change to point counter clockwise. No matter which of the new default rules changes first, a loop is immediately caused for some destination. To handle scenarios like that, we adopts an hybrid approach (Algorithm 2). If the network operators desire some policies that can be guaranteed by existing solutions, e.g., CU or SWAN, such solutions can be specified and plugged in as the fallback mechanism, *FB*. The stream of updates is first handled by GCC's greedy heuristic (Algorithm 1) as long as the policy is preserved. Updates that violate the policy are buffered temporarily. When the buffering time is over a threshold configured by the operator, the fallback mechanism is triggered. The remaining updates are fed into *FB* to be transformed to a feasible sequence, and then Algorithm 1 proceeds with them again to heuristically maximize update parallelism. In that way, GCC avoids getting stuck. Alternatively, if no such external mechanism exists, or the operators highly value update efficiency and ask only for a best effort to maintain consistency during updates, then no fallback would be triggered, i.e., no *FB* would be provided as input to the algorithm. Instead, after a configurable threshold of time, buffered commands are released to the network. That is, operators have a choice on how to balance the trade-off between efficiency and consistency.

To show the feasibility of that approach, we integrated GCC with both CU [23] and SWAN [12] as plugins. The integration with CU is thoroughly evaluated in §7, and here, we focus on the one with SWAN. With SWAN integration, we verified congestion-free invariants of a network shown in Figure 6 (the same topology used in Dionysus [14]). We emulated traffic engineering (TE) and failure recovery two cases (similar to the ones used in Dionysus [14]), and synthesized update orders to preserve the congestion-free property using GCC plus SWAN, and for comparison, with SWAN alone. In the TE case, we changed the network traffic to trigger the TE application to compute new routing updates to match the new traffic allocation. In the failure recovery case, we turned down the link S3-S8 so that link S1-S8 was overloaded. Then the TE application computed new updates to balance the traffic and eventually eliminate the link overload. The detailed events that occurred at all eight switches are depicted in Figure 7. Our hybrid design managed to ensure the same consistency level, but greatly enhanced the parallelism, and thus achieved significant speed improvement (1.95x faster in the TE case, and 1.97x faster in the failure recovery case).

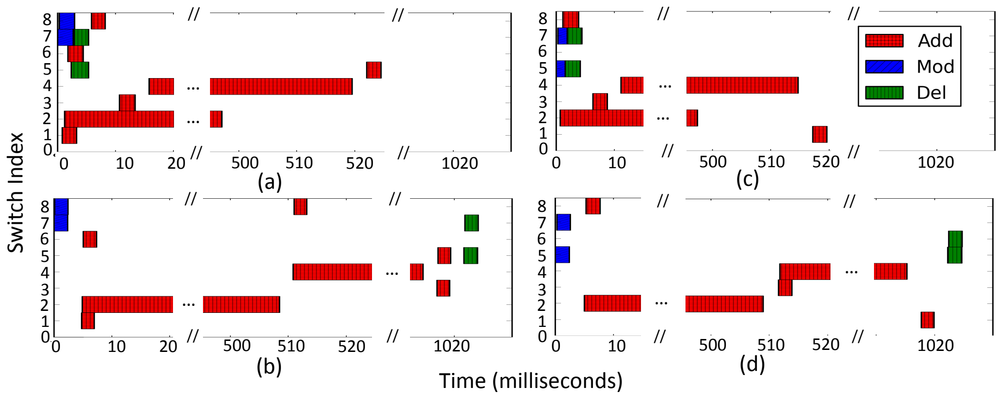


Figure 7: ***Time series of events that occurred across all switches: (a) SWAN + GCC, traffic engineering; (b) SWAN, traffic engineering; (c) SWAN + GCC, failure recovery; (d) SWAN, failure recovery.***

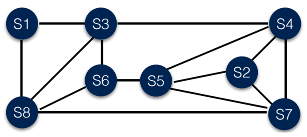


Figure 6: ***Network topology for GCC and SWAN bandwidth experiments***

6 Implementation

We have implemented a prototype of GCC with 8000+ lines of C++ code. GCC is placed between an SDN controller and network devices, intercepting and scheduling network updates issued by the controller in real time. GCC efficiently maintains a group of states, including network-wide data plane rules, uncertainty state of each rule, buffered updates and bandwidth information (e.g., for congestion-free invariants), within a multiple-layer trie in which each layer sub-trie representing a packet header field, as inspired by prior work, VeriFlow [18]. We designed a customized trie data structure for handling different types of rule wildcards, e.g, full wildcard, subnet wildcard, or bismask wildcard [1], differently, and a fast one-pass traversal algorithm to accelerate verification. It's also worth to mention that, besides forwarding rules, the data structure and algorithm are also capable to handle packet transformation rules, such as Network Address Translation (NAT) rules, and rules with VLAN tagging, as later shown in §7. The way we keep track of the uncertainty state of rules enables GCC to detect rules that may cause race conditions, and barrier messages are inserted to serialize them. To bound the amount of time that the controller is uncertain about network states, we implemented two types of the confirmation mechanisms: (1) an application-level acknowledgement by modifying the user-space switch program in Mininet, and (2) leveraging the barrier and barrier reply messages for our physical SDN testbed experiments.

7 Evaluation

7.1 Speed Analysis

We simulated a network consisting of 172 routers following a Rocketfuel topology (AS 1755) [2], and simulated the BGP activities by replaying traces collected from the Route Views Project [3]. After initializing the network with 90,000 BGP updates, we fed 2,559,251 updates into GCC and VeriFlow [18], an existing real-time network verifier. We also varied the number of concurrent uncertain rules in GCC from 100 to 10,000. All of the experiments were performed on a 12-core machine with Intel Core i7 CPU at 3.33 GHz, and 18 GB of RAM, running 64-bit Ubuntu Linux 12.04. The CDFs of the update verification time are shown in Figure 8.

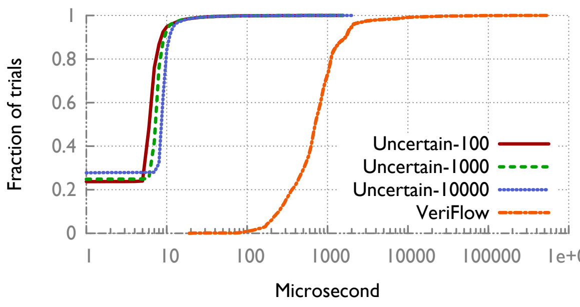


Figure 8: ***Microbenchmark results.***

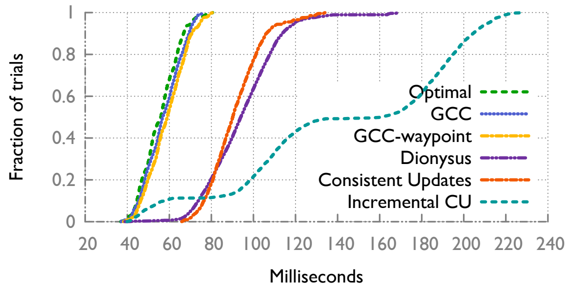
We observed that GCC was able to verify roughly 80% of the updates within 10 μs, and the mean verification time was 9 μs. In fact, approximately 25% of the updates were verified within 1 μs. The reason is that only the minimum change to GCC's network model is required for each update, i.e., only one operation in the trie, as indicated by Figure **??**. GCC verifies the updates much faster than VeriFlow does (by almost two orders of magnitude) mainly because of the data structure optimization (discussed in §6). Both systems exhibit long tail properties, but the verification time of GCC is bounded by 2.16 ms, almost three orders of magnitude faster than VeriFlow's worst case. The results also imply strong system scalability. As the number of concurrent uncertainty rules grows, the verification time in GCC increases slightly (on average, 6.6 μs, 7.3 μs, and 8.2 μs for the 100-, 1000-, and 10000-uncertain-rule cases respectively). Moreover, GCC offers a significant memory overhead reduction relative to VeriFlow: 540 MB versus 9 GB.

7.2 Performance Analysis

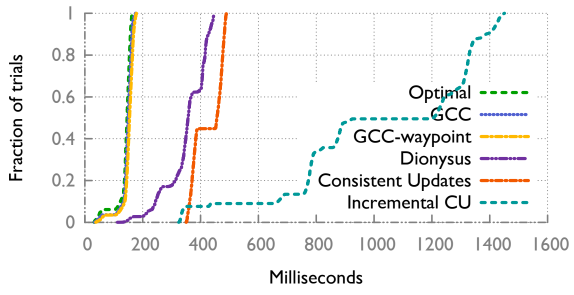
Emulation-based Evaluation: We used Mininet to emulate a fat-tree network with a shortest path routing application and a load-balancing application in a NOX controller. The network consists of five core switches and ten edge switches, and each edge switch connects to five hosts. Once the rules were stable in the switches, we changed the network (e.g., link addition, host migration) to trigger the controller to update the data plane with a set of new updates. For each set of experiments, we tested six update mechanisms: (1) the controller immediately issues updates to the network ("optimal" in terms of update speed); (2) GCC with the basic reachability invariant (loop and black-hole freedom) verification enabled ("GCC"); (3) GCC with an addition invariant that packets must traverse through a specific middle hop before reaching the destination ("GCC-Waypoint"); (4) Consistent Updates ("CU") [23]; (5) incremental Consistent Updates ("Incremental CU") [15]; and (6) Dionysus [14].

We first set the delay between the controller issuing of an update and the corresponding switch finishing of the application of the update (referred to later as the controller-switch delay) to 4 ms to mimic a data center network environment. The settings are in line with that of other data center experiments [7, 24]. We initialized the test with one core switch enabled and added the other four core switches after 10 seconds. The traffic eventually is evenly distributed across all links because of the load balance application. We measured the completion time of updating each communication path, repeated each experiment 10 times, and plotted the CDF for all five scenarios in Figure 9(a).

The performance of GCC (both "GCC" and "GCC-Waypoint") is close to the optimal situation, and takes much less time (around 35 ms reduction on average) than CU. The reason is that with CU, the controller is required to wait for the maximum controller-switch delay to guarantee that all packets can only be handled by either the old or the new rules. GCC relaxes the constraints by allowing a packet being handled by a mixture of old and new rules along the paths, as long as the impact of the new rules passed the verification. By doing so, GCC can apply any verified updates without explicitly waiting for irrelevant updates. CU requires temporary doubling of the storage space for each update, because it does not delete old rules until all in-flight packets processed by the old configuration have drained out of the network. To address that issue, incremental-CU was proposed to trade time against flow table space. By breaking a batch of updates into k subgroups (k was set to 3 in our tests), incremental-CU reduced the extra memory usage to roughly one kth at the cost of multiplying the update time k times. In contrast, when dealing with segment-independent policies, as in this set of experiments, GCC requires no additional memory for network updates, which is particularly useful for many commercial SDN switches that use expensive and power-consuming TCAM memory.



(a) Data center network, transition with link addition



(b) Wide-area network, transition with link addition

Figure 9: ***Emulation results; comparison of update completion time***

SDNs have also been used in wide area networks [12, 13]. To get a sense of wide-area performance, we set the controller-switch delay to 100 ms, and repeated the same tests (Figure 9(b)). GCC requires roughly 200ms less update completion time as compared with CU. The significant improvement is mainly due to the longer controller-switch delay, for which CU and incremental-CU have to wait between the two phases of updates.

We then explored scenarios in which GCC cannot always synthesize a correct update ordering and thus needs to fall back to other solutions to guarantee consistency. The traces we used were collected from a real network of an organization that consists of over 200 layer-3 devices. During a duration of one day (from 16:00 7/22/2014 to 16:00 7/23/2014), we took one snapshot of the network per hour, and used Mininet to emulate 24 transitions, each between two successive snapshots. We processed the network updates with three mechanisms: immediate application of updates, GCC, and CU. The controller-switch delay was set to 4 ms. We selected 10 heavily connected devices in the network, and plotted the number of rules in the network over time during four transition windows, as shown in Figure 10.

We observe that the update completion times (indicated by the width of the span of each curve) using GCC is much shorter (around x% shorter on average with the standard deviation y%) than that using CU, and that the memory needed for storing the rules is much smaller too (around x% smaller on average, with the standard deviation y%). Also, the speed and memory requirements of GCC are close to those of the immediate update case. The reason is that GCC rarely needs to fall back to CU. In 22 out of 24 windows, there were a relatively small number of network updates (around 100+), much as in the [22:00, 23:00) window shown in Figure 10, in which GCC passed through most of the updates with very few fall-backs. During the 23:00 to 1:00 period, there was likely to have been network maintenance, in which 8000+ network updates occurred in each window. Even for such a large number of updates, the number of updates that got stuck and thus forced to a fall-back to CU, was still pretty small (x out of y). Thus we can see, GCC significantly reduces the memory requirement and increases the processing speed. On the other hand, GCC's performance is comparable to that of the immediate update mechanism (the ideal case in terms of speed and memory), but it does not suffer from the short-term network faults that the latter does (e.g., 24 errors in the 0:00 to 2:00 period for the immediate update case). Therefore, in real networking scenarios, GCC is still promising as a way to achieve nearly the "best of both worlds": the efficiency of lightweight update mechanisms in most cases, with the consistency guarantees of more heavyweight solutions.

Physical-testbed-based Evaluation: We also evaluated GCC on a physical SDN testbed with 176 server ports and 676 switch ports, using 13 Pica8 Pronto 3290 switches. We compared the performance of GCC and CU by monitoring the traffic throughput during network transitions. We first created a small-scale network with two sender-receiver pairs transmitting TCP traffic on gigabit links, as shown in Figure 11. Initially, a single link was shared by the pairs, and two flows competed for bandwidth. After 90 seconds, another path was added (the upper portion with dashed lines in Figure 11). Eventually, one flow was migrated to the new path, and each link was saturated. We repeated the experiment 10 times, and recorded the average throughput in a 100-ms window during the network changes. We observed repeatable results and thus plotted the aggregated throughput over time in one trial in Figure 12(a).

GCC took 0.3 second less to finish the transition than CU did. The reasons are that (1) unlike CU, GCC does not require packet modification to support versioning, which takes on the order of microseconds for gigabit links, while the time of packet forwarding is on the order of nanoseconds; (2) CU requires more rule updates and storage than GCC, and the speed of rule installation is around 200 flows per second; and (3) Pica8 OpenFlow switches (with firmware 1.6) cannot simultaneously process rule installations and packets.[2]

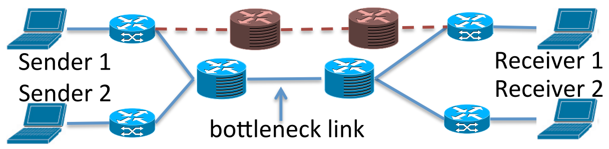


Figure 11: ***Small-scale topology.***

We then created a large topology utilizing all 13 physical SDN switches. We divided each physical switch into 6 "virtual" switches by creating 6 bridges, resulting in 78 switches, and each virtual switch has 8 ports. Initially, the topology consisted of 60 switches, and we randomly selected 10 sender-receiver pairs to transmit TCP traffic. After 90 seconds, we enabled the remaining 18 switches in the network. The topology change triggered installations of new rules being installed for load balancing. We repeated the experiments 10 times, and measured throughput of each flow. We selected two flows from one trial that experienced throughput changes, and show the changes over time in a 100 ms window (Figure 12(b)). The trend of the two flows is consistent with the overall throughput change we observed.

GCC again outperforms CU in terms of convergence time and average throughput during transitions. Compared with CU, GCC spent 20 seconds less to complete the transition (a reduction of around 2/3), because CU needs to wait for the confirmation of all updates in the first phase before proceeding to the second phase. In contrast, GCC's algorithm significantly shortened the delay, especially for networks experiencing a large number of state changes. In addition, the throughput never dropped below 0.9 Gb/s for GCC, while there were temporary yet significant drops for CU during the transition, primarily

[2]All the performance specifications reported in this paper have been confirmed with the Pica8 technical team.

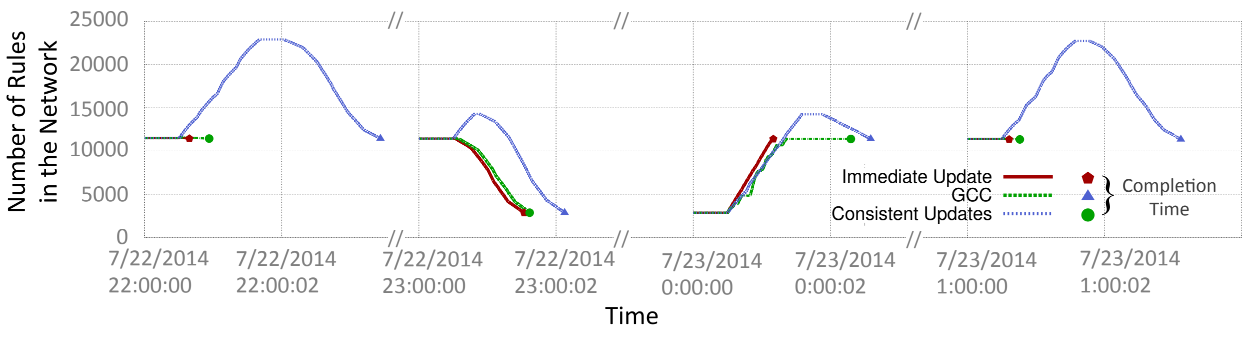
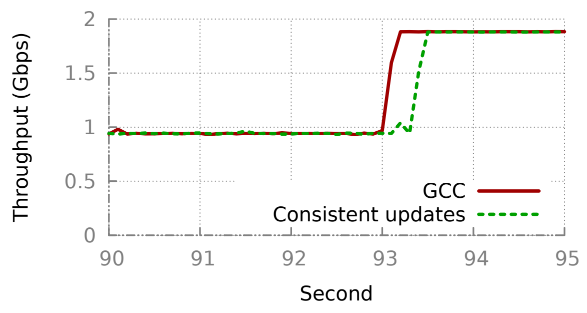
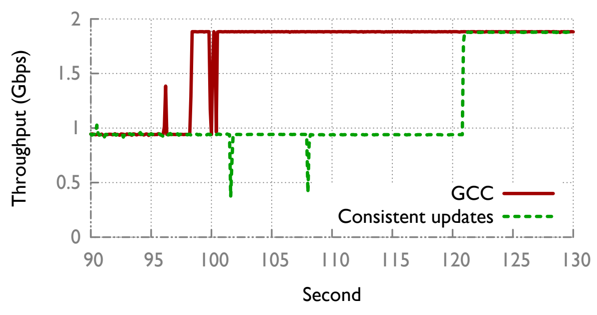


Figure 10: ***Real network trace experiments: (1) Immediate application of updates; (2) GCC (fall back to CU encountered); and (3) Consistent Updates***



(a) A small-scale network with eight switches.



(b) A large-scale network with 78 Switches.

Figure 12: ***Testbed results: comparison of throughput changes during network transitions for Consistent Updates and GCC***

due to the switches' lack of support for simultaneous application of updates and processing of packets.

8 Conclusion

In this paper, we present GCC, a system that enforces customizable network consistency properties with optimized efficiency. We highlight the network uncertainty problem and its effect, and propose a network modelling technique that takes such uncertainty into consideration. The core algorithm of GCC leverages the uncertainty-aware network model, and synthesizes a feasible network update plan (ordering and timing of control messages). In addition to ensuring that there are no violations of consistency requirements, GCC also tries to maximize update parallelism, subject to the constraints imposed by the requirements. Through emulation and experiments on an SDN testbed, we show that GCC is capable of achieving a better consistency vs. efficiency trade-off than existing mechanisms.

A Completeness of the network model

Let us demonstrate that our uncertainty-aware model can accurately capture the view of the network from packets' perspective. We first define the situation when the view of a packet is consistent with the model.

Definition 3. *A packet P's view of the network is* **consistent** *with the uncertainty-aware model, if at any time point during its traversal of the network, the data plane state that the packet encounters is in the model at that time point. More specifically, at time t, to P if a link l*

- *is reachable, l is in the graph model for P at t;*
- *otherwise, l is definitely not certain in the graph at t.*

Theorem 5. *Assuming that no physical failures change the data plane, any packet's view of the network is consistent with the uncertainty-aware model.*

Proof. Without loss of generality, assume that the maximum duration of a packet in the network is δ, which is set as the amount of delay added to confirmations. Consider a packet P that enters the network at time t_1 and leaves at t_2 ($t_2 - t_1 \leq \delta$). Assume that P traverses the network in n hops, and when $n = 0$, P enters the network. Clearly the theorem holds for $n = 0$. Consider hop k, ($k \geq 0$ and $k \leq n$). By induction, at previous hop $(k-1)$, assume that P's view is consistent with the model.

If P encounters a forwarding link at hop k, then there exist two cases. In case 1, the corresponding forwarding rule is intentionally inserted by the controller, and by the time P reaches hop k, the rule is installed. In case 2, the rule is about to be removed, but the action is not done until P has been handled by the rule. Let t_i denote the time of issuing the related command (to add or remove the rule), and t_c the time it is confirmed at the controller as. In case 1, since P reaches the link, $t_i < t$. In the model, that link is modelled as either certain ($t_c \leq t$) or uncertain ($t_c > t$). In case 2, because the link is reachable to P at t, in P's lifetime $[t_1, t_2]$, P's view of the network state contains that link. t_c cannot be earlier than t, because if it were, P could not reach the link. Because of the delayed confirmation mechanism, if an update u causes the removal of the link, the status of the rule remains as uncertain for an extra δ time in the model until $t_c + \delta > t_2$, which is consistent with P's view. In particular, if $t_i \geq t$, then the link is included as certain in the model until the update is issued (t_i).

If P reaches a location where no forwarding rule is available, there are also two cases. In case 1, some forwarding rules have been issued to handle P at this location, but they have not been applied yet. In case 2, there had been available rules, but they were removed before t. In case 1, t_c is definitely later than t. If it weren't, the rule would be there by t. If $t_i < t$, at t, the forwarding rule is only modelled as uncertain. If $t_i \geq t$, at t, the model does not contain that rule. In case 2, the removal of the rules is issued before t. In the interval $[t_i, t_c + \delta]$, any rule R is modeled as uncertain, and after the interval, R is removed from the model, after P leaves the network. Hence, the model is consistent with the view of P during its lifetime in the network. □

References

[1] OpenFlow switch specification. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[2] Rocketfuel: An ISP topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel/.

[3] University of Oregon Route Views Project. http://www.routeviews.org/.

[4] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. *CoNEXT*, 2011.

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[8] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *SIGCOMM*, 2013.

[9] J. Fu, P. Sjodin, and G. Karlsson. Loop-free updates of forwarding tables. *IEEE Transactions on Network and Service Management*, March 2008.

[10] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. *Programming Languages Design and Implementation*, 2013.

[11] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *NSDI*, volume 3, pages 19–21, 2010.

[12] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. *ACM SIGCOMM*, 2013.

[13] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, pages 3–14. ACM, 2013.

[14] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[15] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. *HotSDN*, 2013.

[16] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[17] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[19] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. *ACM SIGCOMM*, 2013.

[20] R. Mahajan and R. Wattenhofer. On consistent updates in software defined networks. *HotNets*, 2013.

[21] A. Noyes, T. Warszawski, P. Černý, and N. Foster. Toward synthesis of network updates. *SYNT*, 2014.

[22] P. Perešíni, M. Kuzniar, N. Vasic, M. Canini, and D. Kostic. OF.CPP: Consistent packet processing for OpenFlow. In *HotSDN*, 2013.

[23] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *ACM SIGCOMM*, 2012.

[24] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? *OSDI*, 2010.

[25] L. Shi, J. Fu, and X. Fu. Loop-free forwarding table updates with minimal link overflow. *International Conference on Communications*, 2009.