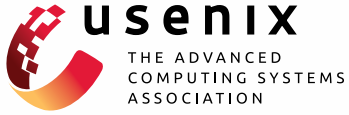


Enforcing Customizable Consistency Properties in Software-Defined Networks

Wenxuan Zhou, *University of Illinois at Urbana-Champaign;*
Dong Jin, *Illinois Institute of Technology;* **Jason Croft, Matthew Caesar, and P. Brighten Godfrey,** *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/zhou

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Enforcing Customizable 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. However, existing techniques impose a high cost in update delay, and/or FIB space. We propose the Customizable Consistency Generator (*CCG*), a fast and generic framework to support *customizable* consistency policies during network updates. *CCG* effectively reduces the task of *synthesizing* an update plan under the constraint of a given consistency policy to a *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 show a large class of consistency policies are guaranteed by this greedy heuristic alone; in addition, *CCG* makes judicious use of existing heavier-weight network update mechanisms to provide guarantees when necessary. As such, *CCG* 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. Mininet and physical testbed evaluations demonstrate *CCG*'s capability to achieve various types of consistency, such as path and bandwidth properties, with zero switch memory overhead and up to a 3× delay reduction compared to previous solutions.

1 Introduction

Network operators often establish a set of correctness conditions to ensure successful operation of the network, such as the preference of one path over another, the prevention of untrusted traffic from entering a secure zone, or loop and black hole avoidance. As networks become an increasingly crucial backbone for critical services, the ability to construct networks that obey correctness criteria is becoming even more important. Moreover, as modern networks are continually changing, it is critical for them to be correct even during transitions. Thus, a key challenge is to guarantee that properties are preserved during transitions from one correct configuration to a new correct configuration, which has been referred as *network consistency* [25].

Several recent proposed systems [13, 16, 19, 25] consistently update software-defined networks (SDNs), transitioning between two operator-specified network snapshots. However, those methods maintain only *specific* properties, and can substantially delay the network update process. Consistent updates [25] (CU), for example, only guarantees *coherence*: during a network update any packet or any flow is processed by either a new or an old configuration, but never by a mix of the two. This is a relatively strong policy that is sufficient to guarantee a large class of more specific policies (no loop, firewall traversal, etc.), but it comes at the cost of requiring a two-phase update mechanism that incurs substantial delay between the two phases and doubles flow entries temporarily. For networks that care only about a weaker consistency property, e.g., only loop freedom, this overhead is unnecessary. At the same time, networks sometimes need properties *beyond* what CU provides: CU only enforces properties on individual flows, but not across flows (e.g., "no more than two flows on a particular link"). SWAN [13] and zUpdate [19] also ensure only a specific property, in their case congestion freedom.

That leads to a question: is it possible to efficiently maintain *customizable* correctness policies as the network evolves? Ideally, we want the "best of both worlds": the efficiency of simply immediately installing updates without delay, but the safety of whatever correctness properties are relevant to the network at hand.

We are not the first to define this goal. Recently, Dionysus [15] proposed to reduce network update time to just what is necessary to satisfy a certain property. However, Dionysus requires a rule 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 waypointing invariant would need a new algorithm. Furthermore, the algorithms work only when forwarding rules match exactly one flow.

We take a different approach that begins with an observation: synthesizing consistent updates for arbitrary consistency policies is hard, but network verification on general policies is comparatively easy, especially now that real-time data plane verification tools [5, 17, 18] can verify very generic data-plane properties of a network state within milliseconds. In fact, as also occurs in domains outside of networking, there is a connection between synthesis and verification. A feasible update sequence is one which the relevant properties are verifiable at each moment in time. Might a verifier serve as a guide through the search space of possible update sequences?

Based on that insight, we propose a new consistent update system, the Customizable Consistency Generator (*CCG*), which efficiently and consistently updates SDNs under customizable properties (invariants), intuitively by converting the scheduling synthesis problem to a series of network verification problems. With *CCG*, network programmers can express desired invariants using an interface (from [18]) which allows invariants to be defined as essentially arbitrary functions of a data plane snapshot, generally requiring only a few tens of lines of code to inspect the network model. Next, *CCG* runs a greedy algorithm: when a new rule arrives from the SDN controller, *CCG* checks whether the network would satisfy the desired invariants if the rule were applied. If so, the rule is sent without delay; otherwise, it is buffered, and at each step *CCG* checks its buffer to see if any rules can be installed safely (via repeated verifications).

This simplistic algorithm has two key problems. First, the greedy algorithm may not find the best (e.g., fastest) update installation sequence, and even worse, it may get stuck with no update being installable without violating an invariant. However, we identify a fairly large scope of policies that are "segment-independent" for which the heuristic is guaranteed to succeed without deadlock (§5.2). For non-segment-independent policies, *CCG* needs a more heavyweight update technique, such as Consistent Updates [25] or SWAN [13], to act as a fallback. But *CCG* triggers this fallback mechanism only when the greedy heuristic determines it cannot offer a feasible update sequence. This is very rare in practice for the invariants we test (§7), and even when the fallback is triggered, only a small part of the transition is left to be handled by it, so the overhead associated with the heavyweight mechanism (e.g., delay and temporarily doubled FIB entries) is avoided as much as possible.

The second challenge lies in the verifier. Existing real-time data plane verifiers, such as VeriFlow and NetPlumber, assume that they have an accurate network-wide snapshot; but the network is a distributed system and we cannot know exactly when updates are applied. To address that, *CCG* explicitly models the uncertainty about network state that arises due to timing, through the use of *uncertain forwarding graph* (§4), a data structure that compactly represents the range of possible network behaviors given the available information. Although compact, *CCG*'s verification engine produces potentially larger models than those of existing tools due to this "uncertainty" awareness. Moreover, as a subroutine of the scheduling procedure, the verification function is called much more frequently than when it is used purely for verification. For these reasons, a substantial amount of work went into optimization, as shown in §7.1.

In summary, our contributions are:

- We developed a system, *CCG*, to efficiently synthesize network update orderings to preserve customizable policies as network states evolve.
- We created a graph-based model to capture network uncertainty, upon which real-time verification is performed (90% of updates verified within 10 μs).
- We evaluate the performance of our *CCG* implementation in both emulation and a physical testbed, and demonstrate that *CCG* offers significant performance improvement over previous work—up to 3× faster updates, typically with zero extra FIB entries—while preserving various levels of consistency.

2 Problem Definition and Related Work

We design *CCG* to achieve the following objectives:

1) Consistency at Every Step. Network changes can occur frequently, triggered by the control applications, changes in traffic load, system upgrades, or even failures. Even in SDNs with a logically centralized controller, the asynchronous and distributed nature implies that no single component can always obtain a fully up-to-date view of the entire system. 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 continuously enforce consistency properties, given the incomplete and uncertain network view at the controller?

2) Customizable Consistency Properties. The range of desired consistency properties of networks is quite diverse. For example, the successful operations of some networks may depend on a set of paths traversing a firewall, certain "classified" hosts being unreachable from external domains, enforcement of access control to protect critical assets, balanced load across links, loop freedom, etc. As argued in [21], a generic framework to handle general properties is needed. Researchers have attempted to ensure certain types of consistency properties, e.g., loop freedom or absence of packet loss [13,19], but those studies do not provide a generalized solution. Dionysus [15], as stated earlier, generalizes the scope of consistency properties it deals with, but still requires designing specific algorithms for different invariants. Consistent Updates [25] is probably the closest solution to support general consistency properties because it provides the relatively strong property of packet coherence which is sufficient to guarantee many other properties; but as we will see next, it sacrifices efficiency.

3) Efficient Update Installation. The network controller should react in a timely fashion to network changes to minimize the duration of performance drops and network errors. There have been proposals [13, 16, 19, 23, 25] that instill correctness according to a specific consistency property, but these approaches suffer substantial performance penalties. For example, the waiting time between phases using the two-phase update

scheme proposed in CU [25] is at least the maximum delay across all the devices, assuming a completely parallel implementation. Dionysus [15] was recently proposed to update networks via dynamic scheduling atop a consistency-preserving dependency graph. However, it requires implementing a new algorithm and dependency graph for each new invariant to achieve good performance. For example, a packet coherence invariant needs one algorithm and a waypoint invariant would need another algorithm. In contrast, our approach reduces the consistency problem to a general network verification problem, which can take a broad range of invariants as inputs. In particular, one only needs to specify the verification function instead of designing a new algorithm. This approach also grants *CCG* the ability to deal with wildcard rules efficiently, in the same way as general verification tools, whereas Dionysus only works for applications with exact match on flows or classes of flows.

3 Overview of *CCG*

CCG converts the update scheduling problem into a network verification problem. Our overall approach is shown in Figure 1. Our uncertainty-aware network model (§4.2) provides a compact symbolic representation of the different possible states the network could be in, providing input for the verification engine. The verification engine is responsible for verifying application updates against specified invariants and policies (§4.4). Based on verification results, *CCG* synthesizes an efficient update plan to preserve policy consistency during network updates, using the basic heuristic and a more heavyweight fallback mechanism as backup (§5.1 and §5.3). One key feature of *CCG* is that it operates in a *black-box* fashion, providing a general platform with a very flexible notion of consistency. For example, one can "plug in" a different verification function and a fallback update scheduling tool to meet one's customized needs.

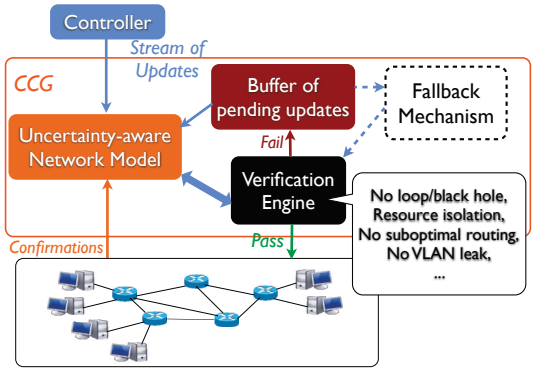


Figure 1: ***System architecture of CCG.***

4 Verification under Uncertainty

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 *uncertain forwarding graphs*, which compactly represent the entire set of possible network states from the standpoint of packets. Next, we describe how we use our model to perform uncertainty-aware network verification (§4.4).

4.1 The Network Uncertainty Problem

Networks must disseminate state among distributed and asynchronous devices, which leads to the inherent *uncertainty* that an observation point has in knowing 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 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 traffic by issuing two instructions in sequence: (1) remove the rule on *A*, and (2) insert a new rule (directing traffic to *A*) on *B*. But it is possible that the second operation finishes earlier than the first one, causing a transient loop that leads to packet losses. That is not an uncommon situation; for example, three out of eleven bugs found by NICE [7] (BUG V, IX and XI) are caused by the control programs' lack of knowledge of the network states.

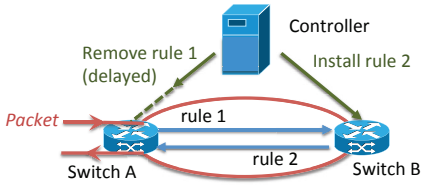


Figure 2: ***Example: challenge of modeling networks in the presence of uncertainty.***

Such errors may have serious consequences. In the previous example, the resulting packet losses could cause a significant performance drop. A recent study [9] shows TCP transfers with loss may take five times longer to complete. Other transient errors could violate security policy, e.g., malicious packets could enter a secure zone because of a temporary access control violation [25].

To make matters worse, errors caused by unawareness of network temporal uncertainty can be permanent. For instance, a control program initially instructs a switch to install one rule, and later removes that rule. The two instructions can be reordered at the switch [11], which ultimately causes 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, this may harm performance because of increasing control traffic and switch operations. There are also scenarios in

which carefully crafting an ordering does not help [25]. In addition, it is difficult for a controller to figure out when to insert the barrier messages. *CCG* addresses that by serializing only updates that have potential to cause race conditions that violate an invariant (§6).

4.2 Uncertainty Model

We first briefly introduce our prior work VeriFlow, a real-time network-wide data plane verifier. VeriFlow intercepts every update issued by the controller before it hits the network and verifies its effect in real time. VeriFlow first slices the set of possible packets into Equivalence Classes (ECs) of packets using all existing forwarding rules and the new update. Each EC is a set of packets that experiences the same forwarding actions throughout the network. Next, VeriFlow builds a *forwarding graph* for each EC affected by the update, by collecting forwarding rules influencing the EC. Lastly, VeriFlow traverses each of these graphs to verify network-wide invariants.

Naively, to model network uncertainty, for every update, we need two graphs to symbolically represent the network behavior with and without the effect of the update for each influenced EC, until the controller is certain about the status of the update. If n updates are concurrently "in flight" from the controller to the network, we would need 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, we efficiently model the network forwarding behavior as a *uncertain forwarding graph*, whose links can be marked as *certain* or *uncertain*. A forwarding link is *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 related forwarding link is changed to *certain*. Such a forwarding graph represents all possible combinations of forwarding decisions at all the devices.

In this way, the extra storage required for uncertainty modeling is linearly bounded by the number of uncertain rules. We next examine when we can resolve uncertainty, either confirming a link as certain or removing it.

4.3 Dynamic 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 forwarding 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 it may result in inconsistencies from the data packets' perspective. Consider a network consisting of four switches, as in Figure 4.

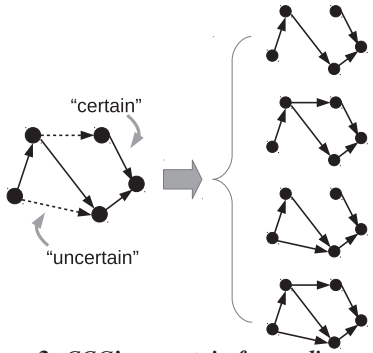


Figure 3: ***CCG's uncertain forwarding graph.***

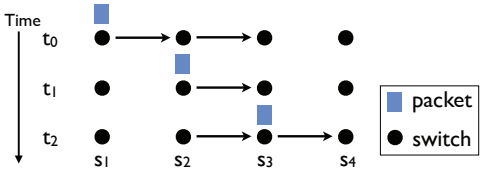


Figure 4: ***Example: challenge of dealing with non-atomicity of packet traversal.***

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, then removes the filtering rule on s_3 at t_2, after the first rule addition has been confirmed.

The forwarding graphs at all steps 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. Traversal of a packet over the network is not atomic, interleaving with network updates, as also observed in [25]. Moreover, [20] recently proved that there are situations where no correct update order exists. To deal with it, upon receiving an ack from the network, *CCG* 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 we want to ensure we represent all the possible behaviors of the network. Even after a forwarding rule has been deleted, packets processed by the rule may still exist in the network, buffered in an output queue of that device, in flight, or on other devices.

We have proved that our uncertainty-aware model is able to accurately capture the view of the network from the packets' perspective [2], even for in-flight packets that have been affected by rules not currently present.

Definition 1. *A packet P's view of the network* **agrees** *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 1. *Assuming that all data plane changes are initiated by the controller, any packet's view of the network agrees with the uncertainty-aware model.*

4.4 Uncertainty-aware Verification

Construction of a *correct* network verification tool is straightforward with our uncertainty-aware model. By traversing the uncertainty graph model using directed graph algorithms, we can answer queries such as whether a reachable path exists between a pair of nodes. That can be done in a manner similar to existing network verification tools like HSA [17] and VeriFlow [18]. However, the traversal process needs to be modified to take into account uncertainty. When traversing an uncertain link, we need to keep track of the fact that downstream inferences lack certainty. If we reach a node with no *certain* outgoing links, it is possible that packets will encounter a black-hole even with multiple *uncertain* outgoing links available. By traversing the graph once, *CCG* can reason about the network state correctly in the presence of uncertainty, determine if an invariant is violated, and output the set of possible conterexamples (e.g., a packet and the forwarding table entries that caused the problem).

5 Consistency under Uncertainty

In this section, we describe how we use our model to efficiently synthesize update sequences that obey a set of provided invariants (§5.1). We then identify a class of invariants that can be guaranteed in this manner (§5.2), and present our technique to preserve consistency for broader types of invariants (§5.3).

5.1 Enforcing Correctness with Greedily Maximized Parallelism

The key goal of our system is to instill user-specified notions of correctness during network transitions. The basic idea is relatively straightforward. We construct a *buffer* of updates received from the application, and attempt to send them out in FIFO order. Before each update is sent, we check with the verification engine on whether there is any possibility, given the uncertainty in network state, that sending it could result in an invariant violation. If so, the update remains buffered until it is safe to be sent.

There are two key problems with this approach. The first is head-of-line blocking: it may be safe to send an update, but one before it in the queue, which isn't safe, could block it. This introduces additional delays in propagating updates. Second, only one update is sent at a time, which is wasteful—if groups of updates do not conflict with each other, they could be sent in parallel.

To address this, *CCG* provides an algorithm for synthesizing update sequences to networks that greedily *maximizes parallelism* while simultaneously obeying the supplied properties (Algorithm 1).

Whenever an update u is issued from the controller, *CCG* 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 $Model$, but marked as uncertain. Otherwise, the update is buffered temporarily in Buf.

When a confirmation of u from the network arrives, *CCG* also intercepts it. The status of u in $Model$ 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.

In this way, *CCG* maintains the order of updates only when it matters. Take the example in Figure 2. If the deletion of rule 1 is issued before the addition of rule 2 is confirmed, *CCG*'s verification engine will capture a possible loop, and thus will buffer the deletion update. Once the confirmation of adding rule 2 arrives, *CCG* checks buffered updates, and finds out that now it's safe to issue the deletion instruction.

5.2 Segment Independence

Next, we identify a class of invariants for which a feasible update ordering exists, and for which *CCG*'s heuristic will be guaranteed to find one such order. As defined in [25], *trace properties* characterize the paths that packets traverse through the network. This covers many common network properties, including reachability, access control, loop freedom, and waypointing. We start with the assumption that a network configuration applies to exactly one equivalence class of packets. A network configuration can be expressed as a set of paths that packets are allowed to take, i.e., a forwarding graph. A configuration transition is equivalent to a transition from an initial forwarding graph, G_0, to a final graph, G_f, through a series of transient graphs, G_t, for $t \in \{1, \ldots, f-1\}$. We assume throughout that the invariant of interest is preserved in G_0 and G_f.

Algorithm 1 Maximizing network update parallelism

ScheduleIndividualUpdate($Model, Buf, u$)

```
On issuing u:
G_uncertain = ExtractGraph(Model, u)
verify = BlackboxVerification(G_uncertain, u)
if verify == PASS then
    Issue u
    Update(Model, u, uncertain)
else
    Buffer u in Buf

On confirming u:
Update(Model, u, certain)
Issue_updates ← ∅
for u_b ∈ Buf do
    G_uncertain = ExtractGraph(Model, u_b)
    verify = BlackboxVerification(G_uncertain, u_b)
    if verify == PASS then
        Remove u_b from Buf
        Update(Model, u_b, uncertain)
        Issue_updates ← Issue_updates + u_b
Issue Issue_updates
```

Loop and black-hole freedom The following theorems were proved for loop freedom [10]: First, given both G_0 and G_f are loop-free, during transition, it is safe (causing no loop) to update a node in any G_t, 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. Second, if there are several updatable nodes in a G_t, any update order among these nodes is loop-free. Third, in any loop-free G_t (including G_0) that is not G_f, there is at least one node safe to update, i.e., a loop-free update order always exists.

Similarly, we have the following proved for the black-hole freedom property [2].

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

Proof. By contradiction. Let N_0, N_1,...N_n be downstream nodes of N_a in G_f. Assume N_0, N_1,...N_n have been updated with respect to G_f in G_t. After updating N_a in G_t, N_0, N_1,...N_n become N_a's downstream nodes and all nodes in the chain from N_a to N_n have been updated. N_a's upstream with respect to G_t can still reach N_a, and thus reach the downstream of N_a. If we assume there is a black-hole from updating N_a, there exists a black-hole in the chain from N_a to N_n. Therefore, the black-hole will exist in G_f, and there is a contradiction. □

Lemma 2. *(Simultaneous updates): Starting with any G_t, any update order among updatable nodes is black-hole-free.*

Proof. Consider a updatable node N_a such that all its downstream nodes in G_f have already been updated in G_t (Lemma 1). Then updating any other updatable node does not change this property. When a node is updatable it remains updatable even after updating other nodes. Therefore, if there are several updatable nodes, they can be updated in any order or simultaneously. □

Theorem 2. *(Existence of a black-hole-free update order): In any black-hole-free G_t that is not G_f (including G_0), at least one of the nodes is updatable, i.e., there is a black-hole-free update order.*

Proof. By contradiction. Assume there is a transient graph G_t such that no node is updatable. All nodes are either updated or not updatable. As nodes with direct links to the destination are updatable (Lemma 1), these nodes can only be updated. Then nodes at previous hop of these nodes in G_t are also updatable (Lemma 1), and therefore these nodes must also be updated. Continuing, it follows that all nodes are updated, which is a contradiction as G_t = G_f . As there is always a node updatable in a consistent G_t, and the updatable node can be updated to form a new consistent G_t, the number of updated nodes will increase. Eventually, all nodes will be updated. Therefore there is a black-hole free update order. □

Any update approved by *CCG* results in a consistent transient graph, so *CCG* always finds a consistent update sequence 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 connectivity between two nodes, both black-hole and loop freedom properties need to hold. Obviously, black-hole freedom is downstream-dependent (Theorem 2), 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. Fortunately, a number of trace properties, such as waypointing, access control, service/middle box chaining, etc., can be broken down to basic connectivity problems. A common characteristic of such properties is that flows are required to traverse a set of waypoints.

Definition 2. Waypoints-based trace property: *A property that specifies that each packet should traverse a set of waypoints (including source and destination) in a particular order.*

Definition 3. Segment dependency: *Suppose a trace property specifies n waypoints, which divide the old and the new flow path each 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 and $(j+1)$ consists of $old_i, old_{i+1}, ..., new_j$, and a loop is formed. As in Figure 5(c), the path could temporarily be $A \rightarrow B \rightarrow 1 \rightarrow B$.

If there is no dependency among segments (Figure 5 (a)), then each can be updated independently simply by ensuring connectivity between the segment's endpoints. That suggests that for paths with no inter-segment dependencies, a property-compliant update order always exists. Another special case is circular dependency between segments, as depicted in Figure 5(d), in which no feasible update order exists.

Theorem 3. *If there is no circular dependency between segments, then an update order that preserves the required property always exists. In particular, if policies are enforcing no more than two waypoints, an update order always exists.*

If a policy introduces no circular dependency, i.e., at least one segment can be updated independently (Figure 5(a-c)), then we say the policy is *segment independent*. 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, forwarding graphs do not share links in many cases. For example, as pointed out in [15], a number of flow-based traffic management applications for the network core (e.g., ElasticTree, MicroTE, B4, SWAN [6, 12–14]), 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. To preserve such properties and waypoint-based trace properties that are not segment independent, we can use other heavyweight techniques as a fallback (see 5.3), such as CU [25]. Besides, there are network properties beyond trace properties, such as congestion freedom, and it has been proven that careful ordering of updates cannot always guarantee congestion freedom [13, 27]. To ensure congestion freedom, one approach is to use other heavyweight tools, such as SWAN [13], as a fallback mechanism that the default heuristic algorithm can trigger only when necessary.

5.3 Synthesis of Consistent Update Schedules

When desired policies do not have the segment-independence property (§5.2), it is possible that some buffered updates (through very rare in our experiments) never pass the verification. 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 counterclockwise. No matter which of the new default rules changes first, a loop is immediately caused for some destination. The loop freedom property is not segment-independent in this case, because each default rule is shared by two equivalence classes (destined to two hosts), which results in conflicting dependencies among forwarding links.

To handle such scenarios, we adopt a 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 *CCG*'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 threshold T, 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, *CCG* can always generate a consistent update sequence, *assuming a fallback mechanism exists which can guarantee the desired invariants*.[1] Note that even with FB triggered, *CCG* achieves better efficiency than using FB alone to update the network, because: 1) in the common case, most of updates are not handled by FB; 2) *CCG* only uses FB to "translate" buffered updates and then heuristically parallelize issuing the output of FB, but doesn't wait explicitly as some FB mechanism does, e.g., the waiting time between two phases in CU.

To show the feasibility of that approach, we implemented both CU [25] (see §7) and SWAN [13] as our fallback mechanisms in *CCG*. We emulated traffic engi-

[1] If no appropriate fallback exists, and the invariant is non-segment-independent, *CCG* can no longer guarantee the invariant. In this case, *CCG* can offer a "best effort" mechanism to maintain consistency during updates by simply releasing buffered updates to the network after a configurable threshold of time. This approach might even be preferable for certain invariants where operators highly value update efficiency; we leave an evaluation to future work.

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

(b) Old path: $A \to B \to 2 \to C$, new path: $A \to 2 \to B \to C$. New AB crosses old BC, so AB depends on BC

(c) Old path: $A \to 1 \to B \to C$, new path: $A \to B \to 1 \to C$. New BC crosses old AB, so BC depends on AB

(d) Old path: $A \to\to 1 \to B \to 2 \to C$, new path: $A \to 2 \to B \to 1 \to C$. 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.*

Algorithm 2 Synthesizing update orderings

ScheduleUpdates($Model, Buf, U, FB, T$**)**
  for $u \in U$ **do**
    ScheduleIndividualUpdate($Model, Buf, u$**)**

  On timeout(T**):**
  $\tilde{U}$ **= Translate(**Buf, FB**)**
  for $u \in \tilde{U}$ **do**
    ScheduleIndividualUpdate($Model, Buf, u$**)**

neering (TE) and failure recovery (FR), similar to Dionysus [15], in the network shown in Figure 6. Network updates were synthesized to preserve congestion-freeness using *CCG* (with SWAN as plug-in), and for comparison, using SWAN alone. In the TE case, we changed the network traffic to trigger new routing updates to match the traffic. In the FR case, we turned down the link S3-S8 so that link S1-S8 was overloaded. Then the FR application computed new updates to balance the traffic. The detailed events that occurred at all eight switches are depicted in Figure 7. We see that *CCG* ensured the same consistency level, but greatly enhanced parallelism, and thus achieved significant speed improvement ($1.95\times$ faster in the TE case, and $1.97\times$ faster in the FR case).

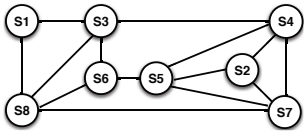


Figure 6: ***Topology for*** *CCG* ***and SWAN bandwidth tests***

6 Implementation

We implemented a prototype of *CCG* with 8000+ lines of C++ code. *CCG* is a shim layer between an SDN controller and network devices, intercepting and scheduling network updates issued by the controller in real time.

CCG maintains several types of state, including network-wide data plane rules, the uncertainty state of each rule, the set of buffered updates, and bandwidth information (e.g., for congestion-free invariants). It stores data plane rules within a multi-layer trie in which each layer's sub-trie represents a packet header field. We designed a customized trie data structure for handling different types of rule wildcards, e.g., full wildcard, subnet wildcard, or bitmask wildcard [24], and a fast one-pass traversal algorithm to accelerate verification. To handle wildcarding for bitmasks, each node in the trie has three child branches, one for each of {0, 1, don't care}. For subnetting, the wildcard branch has no children, but points directly to a next layer sub-trie or a rule set. Thus, unlike other types of trie, the depth of subnet wildcard tries is not fixed as the number of bits in this field, but instead equals to the longest prefix among all the rules it stores. Accordingly, traversal cost is reduced compared with general tries. For full wildcard fields, values can only be non-wildcarded or full wildcarded. The specialized trie structure for this type of field is a plain binary tree plus a wildcard table.

When a new update arrives, we need to determine the set of affected ECs, as well as the rules affecting each EC. VeriFlow [18] performs a similar task via a two-pass algorithm, first traversing the trie to compute a set of ECs, and then for each of the discovered ECs, traversing the trie again to extract related rules. In *CCG*, using callback functions and depth first searching, the modeling work is finished with only one traversal. This algorithm eliminates both the unnecessary extra pass over the trie and the need to allocate memory for intermediate results.

In addition to forwarding rules, the data structure and algorithm are also capable of handling packet transformation rules, such as Network Address Translation (NAT) rules, and rules with VLAN tagging, which are used by CU for versioning, and verified by *CCG* when the CU plug-in is triggered (see §7).

To keep track of the uncertainty states of rules, we design a compact state machine, which enables *CCG* to detect rules that cause potential race conditions. If desired, our implementation can be configured to insert barrier messages to serialize those rule updates.

To bound the amount of time that the controller is uncertain about network states, we implemented two alternate types of the confirmation mechanisms: (1) an application-level acknowledgment by modifying the user-space switch program in Mininet, and (2) leveraging the barrier and barrier reply messages for our physical SDN testbed experiments.

CCG exposes a set of APIs that can be used to write general queries in C++. The APIs allow the network operator to get a list of affected equivalence classes given an arbitrary forwarding rule, the corresponding forwarding graphs, as well as traverse these graphs in a controlled manner and check properties of interest. For instance, an operator can ensure packets from an insecure source

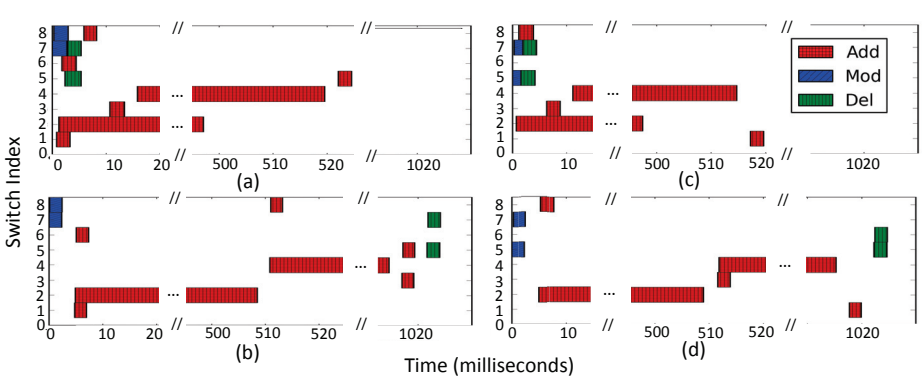


Figure 7: ***Time series of events that occurred across all switches: (a) SWAN + CCG, traffic engineering; (b) SWAN, traffic engineering; (c) SWAN + CCG, failure recovery; (d) SWAN, failure recovery. In both cases, CCG + SWAN finishes about 2x faster.***

encounter a firewall before accessing an internal server.

7 Evaluation

7.1 Verification Time

To gain a baseline understanding of *CCG*'s performance, we micro-benchmarked how long the verification engine takes to verify a single update. We simulated BGP routing changes by replaying traces collected from the Route Views Project [4], on a network consisting of 172 routers following a Rocketfuel topology (AS 1755) [1]. After initializing the network with 90,000 BGP updates, 2,559,251 updates were fed into *CCG* and VeriFlow [18] (as comparison). We also varied the number of concurrent uncertain rules in *CCG* from 100 to 10,000. All 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.

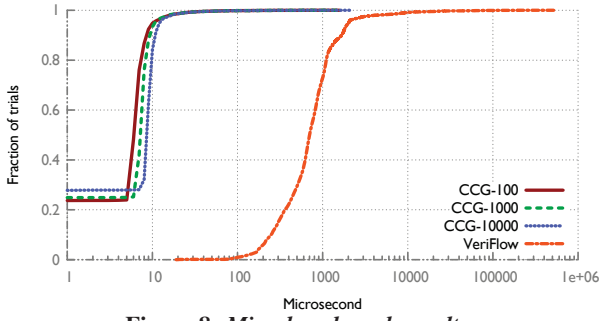


Figure 8: ***Microbenchmark results.***

CCG was able to verify 80% of the updates within 10 μs, with a 9 μs mean. *CCG* verifies updates almost two order of magnitude faster than VeriFlow because of data structure optimizations (§6). Approximately 25% of the updates were processed within 1 μs, because *CCG* accurately tracks the state of each rule over time. When a new update matches the pattern of some existing rule, it's likely only a minimum change to *CCG*'s network model is required (e.g., only one operation in the trie, with no unnecessary verification triggered). We observed long tails in all curves, but the verification time of *CCG* is bounded by 2.16 ms, almost three orders of magnitude faster than VeriFlow's worst case. The results also show strong scalability. As the number of concurrent uncertainty rules grows, the verification time 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, *CCG* offers a significant memory overhead reduction relative to VeriFlow: 540 MB vs 9 GB.

7.2 Update Performance Analysis

7.2.1 Emulation-based Evaluation

Segment-independent Policies: 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. We change the network (e.g., add links, or migrate hosts) 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, which is **Optimal** in terms of update speed; (2) *CCG* with the basic connectivity invariants, loop and black-hole freedom, enabled (*CCG*); (3) *CCG* with an additional invariant that packets must traverse a specific middle hop before reaching the destination (*CCG*-waypoint); (4) Consistent Updates (**CU**) [25]; (5) incremental Consistent Updates (**Incremental CU**) [16]; and (6) **Dionysus** [15] with its WCMP forwarding dependency graph generator. We configure our applications as the same type as in Dionysus, with forwarding rules matching exactly one flow, i.e., no overlapping forwarding graphs. Thus, loop and black-hole freedom are segment-independent as proved in §5.2. Because of the fat-tree structure, there is no crossing between path segments (as in Fig 5(a)), so the waypoint policy is also segment independent. A mix of old and new configurations, e.g., $oldAB + newBC$ in Figure 5(a), is allowed by *CCG*, but forbidden when using CU. Note here, we used our own implementation of the algorithms introduced in Dionysus paper, specifically the algorithm for packet coherence. Therefore, this is not a full evaluation of the Dionysus *approach*: one

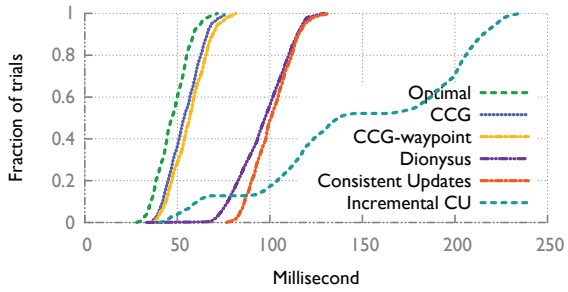
can develop special-purpose algorithms that build customized dependency graphs for weaker properties, and thus achieve better efficiency. We leave such evaluation to future work.

We first set the delay between the controller issuing an update and the corresponding switch finishing the application of the update (i.e, the controller-switch delay) to a normal distribution with 4 ms mean and 3 ms jitter, to mimic a dynamic data center network environment. The settings are in line with that of other data center SDN experiments [8, 26]. 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 balancer application. We measured the completion time of updating each communication path, repeated each experiment 10 times. Figure 9(a) shows the CDFs for all six scenarios.

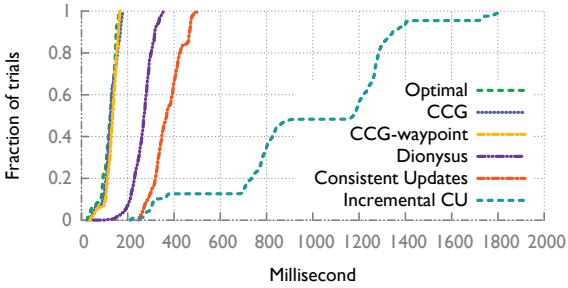
The performance of both "*CCG*" and "*CCG*-waypoint" is close to optimal, and much faster (47 ms reduction on average) than CU. In CU, the controller is required to wait for the maximum controller-switch delay to guarantee that all packets can only be handled by either the old or the new rules. *CCG* 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 verification. By doing so, *CCG* can apply any verified updates without explicitly waiting for irrelevant updates. CU requires temporary doubling of the FIB 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 this, incremental-CU was proposed to trade time against flow table space. By breaking a batch of updates into k subgroups ($k = 3$ 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, *CCG* never needs to trigger any heavyweight fallback plug-in, and thus requires no additional memory, which is particularly useful as switch TCAM memory can be expensive and power-hungry.

To understand how *CCG* performs in wide-area networks, where SDNs have also been used [13, 14], we set the controller-switch delay to 100 ms (normal distribution, with 25ms jitter), and repeated the same tests (Figure 9(b)). *CCG* saved over 200 ms update completion time compared to CU, mainly due to the longer controller-switch delay, for which CU and incremental-CU have to wait between the two phases of updates.

As for Dionysus, we observed in Figure 9 that it speeds up updates compared to CU in both local and wide-area settings, as it reacts to network dynamics rather than pre-determining a schedule. But because its default algorithm for WCMP forwarding produces basically the



(a) Data center network setting



(b) Wide-area network setting

Figure 9: ***Emulation results: update completion time comparison.***

same number of updates as CU, *CCG* (either *CCG* or *CCG*-waypoint) outperforms it in both time and memory cost. We further compared *CCG*-waypoint with Dionysus in other dynamic situations, by varying controller-switch delay distribution. Figure 10 shows the 50^{th}, 90^{th} and 99^{th} percentile update completion time, under various controller-switch delays (normal distributed with different (mean, jitter) pairs, (a,b)) for four update mechanisms: optimal, *CCG*, Dionysus, and CU. In most cases, both *CCG* and Dionysus outperform CU, with one exception (4ms delay, zero jitter). Here, Dionysus does not outperform CU because it adjusts its schedule according to network dynamics, which was almost absent in this scenario. The cost of updating dependency graphs in this scenario is relatively large compared to the small network delay. When the mean delay was larger (100ms), even with no jitter, Dionysus managed to speed the transition by updating each forwarding path independently. On the other hand, *CCG*'s performance is closer to Optimal than Dionysus. For example, in the $(4,0)$ case, *CCG* is 37%, 38%, and 52% faster than Dionysus in the 50^{th}, 90^{th} and 99^{th} percentile, respectively; in the $(100,25)$ case, *CCG* is 50%, 50%, and 53% faster than Dionysus in the 50^{th}, 90^{th} and 99^{th} percentile, respectively. Also, we observe that Dionysus's performance is highly dependent on the variance of the controller-switch delay (the larger the jitter is, the faster the update speed) because of the dynamic scheduling, but *CCG*'s performance is insensitive to the jitter.

Non-segment-independent Policies: We then explored

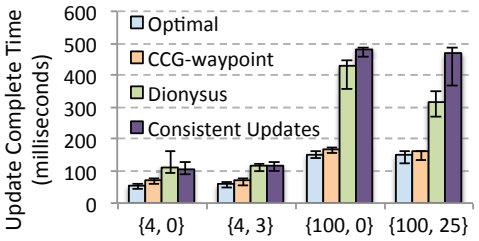


Figure 10: ***Update completion time with [50^{th}, 90^{th}, 99^{th} percentile]; x-axis label {a, b}: a is the mean controller-switch delay, b is the jitter following a normal distribution.***

scenarios in which *CCG*'s lightweight heuristic cannot always synthesize a correct update ordering and needs to fall back to the more heavyweight algorithm to guarantee consistency. The traces we used were collected from a relatively large enterprise network that consists of over 200 layer-3 devices. During a one-day period (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, *CCG*, and CU. Updates were issued such that new rules were added first, then old rules deleted. Thus, all three mechanisms experience the trend that the number of stored rules increases then decreases.. The controller-switch delay was set to 4 ms. We selected 10 strongly connected devices in the network, and plotted the number of rules in the network over time during four transition windows, as shown in Figure 11. As the collected rules overlapped with longest prefix match, the resulting forwarding graphs might share links, so unlike previous experiments, segment-independency was not guaranteed.

The update completion time (indicated by the width of the span of each curve) using *CCG* was much shorter than CU, and the memory needed to store the rules was much smaller. In fact, the speed and memory requirements of *CCG* were close to those of the immediate update case, because *CCG* rarely needs to fall back to CU. In 22 out of 24 windows, there was a relatively small number of network updates (around 100+), much as in the [22:00, 23:00) window shown in Figure 11, in which *CCG* passed through most of the updates with very few fallbacks. During the period 23:00 to 1:00, there was a burst of network dynamics (likely to have been caused by network maintenance), in which 8000+ network updates occurred. Even for such a large number of updates, the number of updates forced to a fallback to CU, was still quite small (10+). Since *CCG* only schedules updates in a heuristic way, the waiting time of a buffered update could be suboptimal, as in this hour's case, where the final completion time of *CCG* was closer to CU. *CCG* achieves performance comparable to the immediate update mechanism, but without any of its short-term network faults (24 errors in the 0:00 to 2:00 period).

7.2.2 Physical-testbed-based Evaluation

We also evaluated *CCG* on a physical SDN testbed [3] consisting of 176 server ports and 676 switch ports, using Pica8 Pronto 3290 switches via TAM Networks, NIAGARA 32066 NICs from Interface Masters, and servers from Dell. We compared the performance of *CCG* and CU by monitoring the traffic throughput during network transitions. We first created a network with two sender-receiver pairs transmitting TCP traffic on gigabit links, shown in Figure 12. 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 12). 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. Figure 13(a) shows the aggregated throughput over time for one trial.

CCG took 0.3 seconds less to finish the transition than CU because: (1) unlike CU, *CCG* does not require packet modification to support versioning, which takes on the order of microseconds for gigabit links, while packet forwarding is on the order of nanoseconds; (2) CU requires more rule updates and storage than *CCG*, 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]

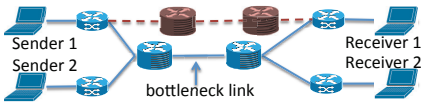


Figure 12: ***eight-switch topology.***

To test *CCG* in a larger setting, we then utilized all 13 physical switches. Each physical switch was devided into 6 "virtual" switches by creating 6 bridges. Due to the fact that the switches are physically randomly connected, this division results in a "pseudo-random" network consisting of 78 switches, each with 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 to balance load. We repeated the experiments 10 times, and selected two flows from one trial that experienced throughput changes (Figure 13(b)). The trend of the two flows is consistent with the overall observed throughput change.

CCG again outperformed CU in convergence time and average throughput during transitions. Compared to CU, *CCG* spent 20 fewer seconds to complete the transition (a reduction of 2/3), because CU waits for confirmation

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

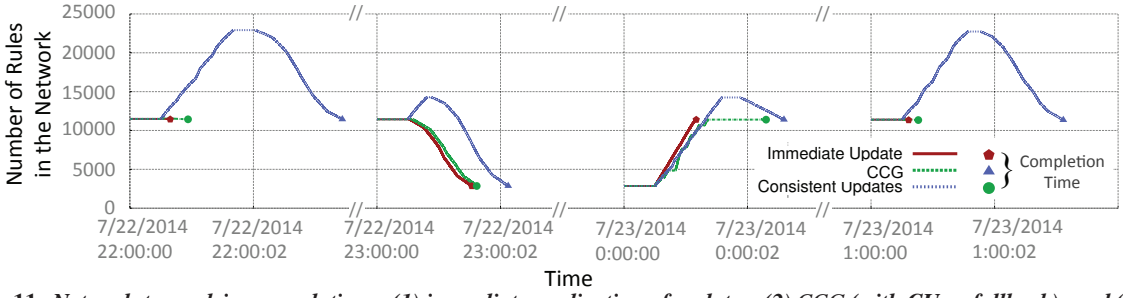
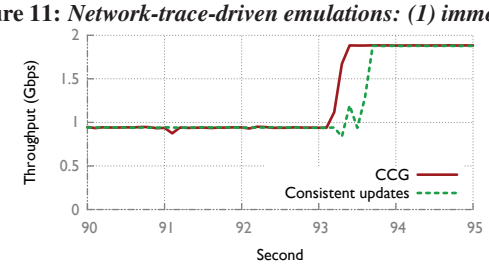
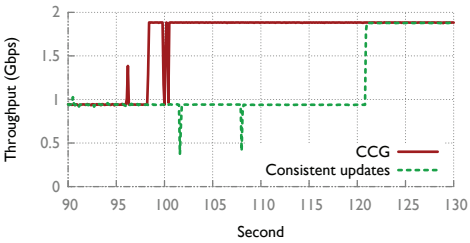


Figure 11: ***Network-trace-driven emulations: (1) immediate application of updates; (2) CCG (with CU as fallback); and (3) CU.***



(a) A eight-switch topology.



(b) A 78-switch network.

Figure 13: ***Physical testbed results: comparison of throughput changes during network transitions for CCG and CU.***

of all updates in the first phase before proceeding to the second. In contrast, *CCG*'s algorithm significantly shortened the delay, especially for networks experiencing a large number of state changes. In *CCG*, the throughput never dropped below 0.9 Gb/s, while CU experienced temporary yet significant drops during the transition, primarily due to the switches' lack of support for simultaneous application of updates and processing of packets.

8 Discussion

Limitations: *CCG* synthesizes network updates with only heuristically maximized parallelism, and in the cases where required properties are not *segment independent*, relies on heavier weight fallback mechanisms to guarantee consistency. When two or more updates have circular dependencies with respect to the consistency properties, fallback will be triggered. One safe way of using *CCG* is to provide it with a strong fallback plug-in, e.g., CU [25]. Any weaker properties will be automatically ensured by *CCG*, with fallback triggered (rare in practice) only for a subset of updates and when necessary. In fact, one can use *CCG* even when fallback is always on. In this case, *CCG* will be faster most of the time, as discussed in §5.3.

Related work: Among the related approaches, four warrant further discussion. Most closely related to our work is Dionysus [15], a dependency-graph based approach that achieves a goal similar to ours. As discussed in §2, our approach has the ability to support 1) flexible properties with high efficiency without the need to implement new algorithms, and 2) applications with wildcarded rules. [22] also plans updates in advance, but using model checking. It, however, does not account for the unpredictable time switches take to perform updates. In our implementation, CU [25] and VeriFlow [18] are chosen as the fallback mechanism and verification engine. Nevertheless, they are replaceable components of the design. For instance, when congestion freedom is the property of interest, we can replace CU with SWAN [13].

Future work: We plan to study the generality of *segment independent* properties both theoretically and practically, test *CCG* with more data traces, and extend its model to handle changes initiated from the network. As comparison, we will test *CCG* against the original implementation of Dionysus with dependency graphs customized to properties of interest. We will also investigate utilizing possible primitives in network hardware to facilitate consistent updates.

9 Conclusion

We present *CCG*, a system that enforces customizable network consistency properties with high efficiency. We highlight the network uncertainty problem and its ramifications, and propose a network modeling technique correctly derives consistent outputs even in the presence of uncertainty. The core algorithm of *CCG* 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, *CCG* also tries to maximize update parallelism, subject to the constraints imposed by the requirements. Through emulations and experiments on an SDN testbed, we show that *CCG* is capable of achieving a better consistency vs. efficiency trade-off than existing mechanisms.

We thank our shepherd, Katerina Argyraki, for helpful comments, and the support of the Maryland Procurement Office under Contract No. H98230-14-C-0141.

References

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

[2] Tech report. http://web.engr.illinois.edu/~wzhou10/gcc_tr.pdf.

[3] University of illinois ocean testbed. http://ocean.cs.illinois.edu/.

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

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

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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

[22] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. *Programming Languages Design and Implementation*, 2015. to appear.

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

[24] Open Network Foundation. OpenFlow switch specification v1.4, October 2013. https://www.opennetworking.org/.

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

[26] 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.

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