Incremental Consistent Updates

Naga Praveen Katta
Princeton University
nkatta@cs.princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

ABSTRACT

A *consistent update* installs a new packet-forwarding policy across the switches of a software-defined network in place of an old policy. While doing so, such an update guarantees that every packet entering the network either obeys the old policy or the new one, but not some combination of the two. In this paper, we introduce new algorithms that trade the *time* required to perform a consistent update against the *rule-space* overhead required to implement it. We break an update in to k rounds that each transfer part of the traffic to the new configuration. The more rounds used, the slower the update, but the smaller the rule-space overhead. To ensure consistency, our algorithm analyzes the dependencies between rules in the old and new policies to determine which rules to add and remove on each round. In addition, we show how to optimize rule space used by representing the minimization problem as a mixed integer linear program. Moreover, to ensure the largest flows are moved first, while using rule space efficiently, we extend the mixed integer linear program with additional constraints. Our initial experiments show that a 6-round, optimized incremental update decreases rule space overhead from 100% to less than 10%. Moreover, if we cap the maximum rule-space overhead at 5% and assume the traffic flow volume follows Zipf's law, we find that 80% of the traffic may be transferred to the new policy in the first round and 99% in the first 3 rounds.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*

Keywords

Consistent Network Updates, Software-Defined Networking, OpenFlow, Network Programming Languages, Frenetic.

HotSDN'13, August 16, 2013, Hong Kong, China.

1. INTRODUCTION

Network forwarding policies are volatile: They change when traffic load shifts, when new security vulnerabilities are found, and when equipment goes down for maintenance. Software-Defined Networking (SDN) allows network administrators to write their own custom algorithms that compute new network policies quickly and effectively. But during changes in policies, administrators want the flow of traffic to continue satisfying key invariants, like avoiding forwarding loops, blackholes, and violations of access-control policies.

A *consistent update* [8] is a powerful mechanism for SDN that helps ensure key invariants hold during the transitions between policies. More specifically, a consistent update from policy A to policy B ensures that every packet traversing the network is forwarded exclusively according to A or exclusively according to B—not by some other policy. Consistent updates make it easier to reason about network semantics, because any invariant that holds for both A and B also holds across the update.

The generic way to implement a consistent update from a previously-installed policy A to a new policy B uses a *two-phase commit* [8]. To prepare for a two-phase commit, each policy is preprocessed into one that tags all packets entering the network with a version number (*i.e.*, A and B tagging packets with version numbers 1 and 2, respectively). In the core of the network, every rule of the preprocessed policy tests incoming packets for whether they contain the correct version number and, if so, the policy acts as the unpreprocessed policy. As the packets leave the network, the egress switches strip the version numbers. Now, installation of B occurs as follows: (i) install B's rules in the middle of the network, (ii) wait until step 1 completes on all switches, (iii) install B's rules at the network ingress points, (iv) wait until all packets being processed by policy A drain from the network, and (v) remove all of A's rules.

This algorithm is both *safe* (updates are provably consistent) and *general* (it can update between any pair of policies). However, the algorithm has a significant practical drawback: in the period between steps (iii) and (v), switches hold the rules from both policies—requiring *double* the rule space. This rule space, especially TCAM, is expensive and power-hungry, and there is a long literature of optimization techniques designed to minimize rule space utilization (See [6] for a recent example). Reitblatt *et al.* discuss the possibility of saving space by not updating the parts of the network where the policy does not change [8]; however, when policies differ, the required rule space still doubles. Indeed, McGeer [7] observed this problem immedi-

ately. However, McGeer's solution redirects traffic through the controller, introducing substantial overhead and limiting packet-processing throughput. Hong et al. [3] focus on a different approach in which they move the network through a series of intermediate configurations that preserve both rule space and bandwidth constraints. As these intermediate configurations are neither precisely the new or old policy (i.e., their update satisfies slightly weaker properties than a consistent update for more general policies), they solve an important though different problem. In summary, we still need algorithms for consistent network update that adhere to switch resource constraints.

Since TCAM space is a major cost, we believe it is important to ask if one can trade time, or other resources, for TCAM space. Trading time is especially promising when updates are planned in advance (*e.g.*, maintenance), predictable (*e.g.*, known vacillations in load) or delay tolerant (*e.g.*, new nodes/jobs in a data center). In this paper, we introduce a generic algorithm for implementing consistent updates that trades *update time* for *rule-space overhead*. The algorithm reduces the rule-space costs without diverting traffic to the controller so that there is no significant packet-processing delay. We divide a global policy into a set of *consistent slices* and shift to the new policy one slice at a time. By increasing the number of slices, we substantially reduce the rule-space overhead on the switches. Moreover, we demonstrate how to represent the optimal selection and ordering of slices as a mixed integer linear programming problem. In addition, we show how to adjust the algorithm to manage hard rule-space caps and, when given flow volume statistics, to migrate high-volume flows early while minimizing rule space used. We have implemented our algorithm on the NOX controller platform [1] and present preliminary empirical results. These results show, for example, that a 6-round incremental update can reduce space overhead required from 100% to 10%. When flow volume follows Zipf's law (an exponentially decaying distribution observed in many practical scenarios) and rule space overhead is capped at 5%, we transfer almost 99% of traffic to the new policy, in just 3 rounds.

2. INCREMENTAL UPDATE ALGORITHM

The key to managing rule space during a consistent update is to split the update into rounds that each move some fraction of the traffic to the new policy. Once all flows have moved, the global update is complete. During the transition, some fraction of the traffic follows the new policy and some fraction follows the old policy. This is typical of any consistent update mechanism (or any mechanism at all, as simultaneous update of switches is impossible) and does not violate the desired consistency properties. In particular, if both the old and new policies satisfy some invariant of their paths (such as loop freedom or connectivity), all packets traversing the network follow paths with that property.

Intuitively, each round proceeds as follows: (i) choose some subset of the flows entering the network to move from old to new policy, (ii) determine the new rules that must be placed in the network to preserve consistency prior to moving the flows, (iii) install the new rules required using a two-phase commit, (iv) determine the old rules that may be safely removed, while still preserving consistency, (v) wait until any lingering packets from the old policy have drained out of the network, and (vi) remove the old rules allowed.

Notice that step (ii) requires an analysis of the new policy, while step (iv) requires a related analysis of the old policy. The analysis is necessary because the policy rules affecting a flow are not simply a collection of rules on each switch that exactly match that flow. For example, as shown in Figure 1, the packets in a flow may undergo header modifications and hence may match various rules at various points in the network. Also, two flows may match the same old rule (with a compressed ternary match) and hence the rule cannot be removed until both flows migrate to the new policy.

The following paragraphs describe the structure of policies, the analyses that must be performed on them, and a generic incremental update algorithm in more detail.

Policies. A global network *policy* R is a set of *rules* r, and every rule is a triple $\langle P, a, z\rangle$ consisting of a predicate P, an action a, and a priority z. A predicate defines a set of packets, based on the packet header fields and the packet location (the switch and port at which the packet has arrived). For example, the predicate ($\texttt{srcIP} = \texttt{10.0.0.1} \wedge \texttt{switch} = \texttt{3} \wedge \texttt{inport} = \texttt{7}$) defines the set of packets with source IP address `10.0.0.1` located at inport `7` on switch 3. We will typically write $p \in P$ to indicate a packet p matches predicate P or is included in the set defined by P. The predicate for a particular rule can only match packets incoming to one switch (*i.e.*, each rule is local to a particular switch). However, our algorithms process predicates that describe packets located at many switches in the network.

Actions transform packets and/or their location. We leave the possible set of actions abstract in this paper, but they may include any actions supported by OpenFlow, including forwarding packets out single outports, dropping packets, flooding or broadcasting packets out multiple outports, or modifying the fields of packets.

To determine the effect of a policy on a packet, one finds the highest-priority rule that matches the packet and performs the associated actions. We assume policies are *unambiguous*: For any packet, at any location in the network, at most one rule with highest priority matches the packet. Ambiguous policies are easily made unambiguous by elevating the priority of one or more rules. We also assume policies are *versioned*. A *k-version policy* (1) modifies all packets entering the network so they are tagged with version number k,[1] (2) only matches on packets tagged k in the middle of the network, and (3) strips tag k off of packets at the network egress. Such policies are also assumed to be *ingress total*; that is, there should exist a rule that processes (and tags with k) every packet entering the network.

Policy Analysis. In each round i of the update, one chooses a predicate, P_i, that describes a subset of the packets entering a network. This predicate defines the flows that move to the new policy on this round. Any such predicate P_i over ingress packets defines what we call a *slice* of a policy. Given an ingress predicate P, a policy R, and a topology T, the slice defined by P in R is the set of rules reachable by packets p that originate in the set defined by P and that flow through the network defined by the policy R and the topology T.

Now, given the predicate P_i for round i, one must compute the slice defined by P_i through the *new policy*. This slice

[1]VLAN tags or MPLS labels may be used for tagging.

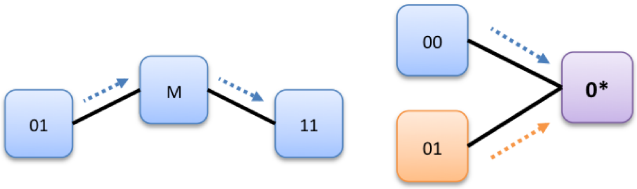

Figure 1: (a) Header modifications at a middlebox, (b) Multiple predicates reaching the same rule

```
// packets reachable from P in T through R
procedure reachable_packets(P:Pred, R:Policy, T:Topo){
  reaches = P;
  while (eval(R,T,reaches) ∪ reaches) != reaches do
    reaches = reaches ∪ eval(R,T,reaches));
  return reaches;
}

// rules in R reachable from ingress predicate P
procedure reachable_rules(P:Pred, R:Policy, T:Topo) {
  packets = reachable_packets (P, R, T);
  rules = ∅;

  for each r in R in descending rule priority order
    if (r.predicate ∩ packets) != ∅ then
      rules = rules  ∪ {r};
      packets = packets - r.predicate;

  return rules;
}
```

Figure 2: The slice generation algorithm.

includes all rules that must be added to support the flow of the P_i-traffic through the network. If any rule in this set is omitted, some packet is unable to complete its journey across the network properly. Omitting such a rule causes this round of the update to violate the per-packet consistency property.

Next, suppose the slice of the new policy corresponding to the disjunction of predicates $P_1 \vee \cdots \vee P_i$ has been installed. Now, one needs to know which *old* rules must remain to process the traffic deferred to later rounds. To identify these rules, we compute the slice generated by the complementary ingress predicate through the old policy, *i.e.*, $\neg(P_1 \vee \cdots \vee P_i) \wedge \mathsf{Ingress}$, where Ingress defines the complete set of ingress packets.

Figure 2 presents the slice-generation algorithm, which may be viewed as a form of header-space (reachability) analysis [5]. The function `eval(R,T,P)` returns the new set of packets obtained by evaluating the policy `R` over the input packet set `P` and then pushing the resultant packets across the network topology `T` to their new locations.

Generic Update Algorithm. Figure 3 presents a complete, generic algorithm for incremental consistent update. The main algorithm updates the network from the `old` policy to the `new` policy under topology `T`. The variable `old_remains` tracks the set of rules from the old policy yet to be evicted from the network; it is initially equal to `old`. The variable `new_placed` tracks the set of rules from the new policy already installed in the network; it is initially equal to the empty set, since no new rules have been installed. The variable `packets` tracks the flows that remain to be migrated from `old` to `new` policies; it is initially `ingress`—the complete set of packets that may enter the network.

```
procedure update(old:Policy, new:Policy, T:Topo) {
  old_remains = old;
  new_placed  = ∅;
  packets     = ingress;

  while packets != ∅ do
    choose ingress predicate P;

    // compute new rules to be added to switches
    to_add = reachable_rules(P, new, T)-new_placed;
    new_placed = new_placed ∪ to_add;

    // compute the old rules to be removed
    packets = packets - P;
    required = reachable_rules (packets, old, T);
    to_remove = old_remains - required;
    old_remains = required;

    // perform the slice update consistently
    two_phase_commit(to_add, to_remove);

  return;
}
```

Figure 3: Incremental consistent update alg.

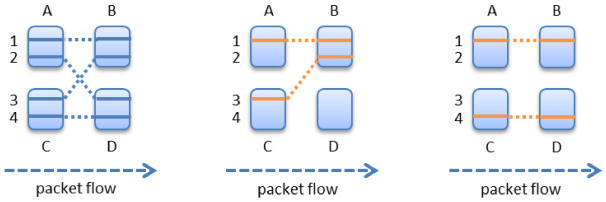

Figure 4: The effect of predicate choice.

Each iteration of the while loop corresponds to a round of the update procedure. At the beginning of the round, we choose `P`, a predicate defining the new flows to migrate from old to new policies. For now, we leave the choice abstract—any series of choices defines a correct update. In the middle of the loop, we compute the new rules to be added and the old rules to be removed, using the policy analysis described earlier. Then, we perform a two-phase update [8], and repeat until the set of packets left to move becomes empty.

3. RULE SPACE OPTIMIZATION

The algorithm in Figure 3 does not describe how to choose a series of predicates that minimize the rule-space overhead, given a fixed number of rounds for an update. To understand why ingress predicate selection is challenging, consider the diagrams in Figure 4.

The left side shows a network with switches labeled A, B, C, and D, and a new policy we want to install. The new policy dictates that, eventually, each switch will contain two rules, as shown by the dark (blue) lines. The rules labelled 1, 2, 3, and 4 on switches A and C, respectively handle ingress packets. Packets flow from one rule to the next as indicated by the dotted lines.

The central diagram demonstrates what happens if we choose an ingress predicate that overlaps with rules 1 and 3 for the first round of the update. Since packets flow from rules 1 and 3 to both rules on B, we are forced to add both rules of B at once during round 1. If instead, we had chosen a predicate that overlaps with rules 1 and 4, as in the rightmost diagram, we would avoid overloading switch B on the first round of the update.

Constants	
K	number of rounds
S	number of switches
R^{old}	old policy
R^{new}	new policy
$R^{old}_{s,m}$	mth rule in the old policy on switch s
$R^{new}_{s,m}$	mth rule in the new policy on switch s
M^{old}_s	number of rules in old policy on switch s
M^{new}_s	number of rules in new policy on switch s
M^{max}_s	$\max(M^{new}_s, M^{old}_s)$
P_n	an ingress predicate ($n \in [1..N]$)
$W^{new}_{n,s,m}$	weight of the rule $R^{new}_{s,m}$ for predicate P_n
$W^{old}_{n,s,m}$	weight of the rule $R^{old}_{s,m}$ for predicate P_n
T_s	rule-space overhead threshold/cap on switch s
Variables	
O	Rule space overhead (to be minimized)
$X_{n,k}$	1 if P_n is selected by end of round k
$Y^{new}_{k,s,m}$	1 if $R^{new}_{s,m}$ is added to s by end of round k
$Y^{old}_{k,s,m}$	1 if $R^{old}_{s,m}$ is deleted from s by end of round k
$Z^{new}_{k,s}$	total new rules added to s by end of round k
$Z^{old}_{k,s}$	total number old rules removed from s by the end of round k

Figure 5: Optimization Variables and Constants

MIP Setup. To optimize predicate selection and ordering, we have represented the problem as a mixed integer linear program (MIP). Figure 5 defines the set of constants and variables that appear in the MIP. Of particular interest is the set of predicates $P_1, \ldots, P_N$. The goal of the algorithm is to select some subset of those predicates each round so that the overhead (O) is minimized. Variable $X_{n,k}$ is 1 if P_n is selected by round k and is 0 otherwise. Thus the algorithm must be seeded with a specific set of predicates $P_1, \ldots, P_N$, which will then be assigned to rounds.

We seed our algorithm with the set formed by the union of predicates from all rules on all switches in the new policy. But this set of predicates can be any arbitrary set that completely defines the flow space of packets entering the network. For example, the collection of predicates from the (old/new) rules on just the ingress switches is also a good choice. The more the predicates added to this set, the better the optimal solution, because of a wider choice of slice selection in each round, at the cost of more computation.

The constant $W^{old}_{n,s,m}$ gives the *weight* of the rule $R_{s,m}$ for the predicate P_n in the old policy, which is determined by whether a packet matching P_n can flow to $R_{s,m}$. If no packet can flow from P_n to $R^{old}_{s,m}$, then $W^{old}_{n,s,m}$ is 0. If, on the other hand, packets can flow from multiple predicates $P_1, \ldots, P_j$ (not necessarily the first j predicates in the order) to $R^{old}_{s,m}$ (and from no other predicate), then the weights $W^{old}_{1,s,m}, \ldots, W^{old}_{j,s,m}$ would each be equal to $1/j$; the weights $W^{new}_{n,s,m}$ for the new policy are calculated analogously. Using the fraction $1/j$ ensures that, if new rules corresponding to predicates $P_1, \ldots, P_j$ have been installed and the weights $W^{old}_{1,s,m}, \ldots, W^{old}_{j,s,m}$ sum to 1, then the rule $R^{old}_{s,m}$ is no longer needed—no packets following the old policy can flow to that rule any longer. Instead, all packets that could have flowed to that rule are now handled by the new policy.

MIP Equations. Figure 6 presents the set of equations that define the MIP. Equations 2 and 3 give the basic constraints

$$\min\ O \text{ subject to} \tag{1}$$

$$X_{n,k+1} \geq X_{n,k} \quad \forall n \in [N], k \in [K-1] \tag{2}$$

$$X_{n,K} = 1 \quad \forall n \in [N] \tag{3}$$

$$Y^{new}_{k,s,m} \geq \sum_{n=1}^{N} X_{n,k} W^{new}_{n,s,m} \quad \forall k \in [K], s \in [S], m \in [M^{new}_s] \tag{4}$$

$$Y^{old}_{k,s,m} \leq \sum_{n=1}^{N} X_{n,k} W^{old}_{n,s,m} \quad \forall k \in [K], s \in [S], m \in [M^{old}_s] \tag{5}$$

$$Z^{new}_{k,s} = \sum_{m=1}^{M^{new}_s} Y^{new}_{k,s,m} \quad \forall k \in [K], s \in [S] \tag{6}$$

$$Z^{old}_{k,s} = \sum_{m=1}^{M^{old}_s} Y^{old}_{k,s,m} \quad \forall k \in [K], s \in [S] \tag{7}$$

$$O \geq \frac{(M^{old}_s + Z^{new}_{k+1,s} - Z^{old}_{k,s}) - M^{max}_s}{M^{max}_s} \quad \forall k \in [K-1], s \in [S] \tag{8}$$

$$\begin{aligned} &X_{n,k} \in \{0,1\} && \forall k \in [K], n \in [N] \\ &Y^{new}_{k,s,m}, Y^{old}_{k,s,m} \in \{0,1\} && \forall k \in [K], s \in [S], m \in [M(s)] \\ &Z^{new}_{k,s}, Z^{old}_{k,s}, O \in \mathbb{R}^+ && \forall k \in [K], s \in [S] \end{aligned} \tag{9}$$

Figure 6: The rule space optimization problem.

on the variables $X_{n,k}$, which determine whether predicate P_n is selected by the kth round. Equation 3 guarantees that all predicates are selected within K rounds.

Equation 4 constrains $Y^{new}_{k,s,m}$, which defines whether or not the rule $R^{new}_{s,m}$ must be added to the switch by the end of this round k. Since $Y^{new}_{k,s,m}$ is 0 or 1, this equation states that if there is *any* non-zero weight on $R^{new}_{s,m}$, then it must be added. Dually, equation 5 constrains $Y^{old}_{k,s,m}$, which defines whether or not the rule $R_{s,m}$ should be deleted on the switch s by the end of round k. Notice the opposite orientation of the inequality here. Since $Y^{old}_{k,s,m}$ is also 0 or 1, this rule states that $R^{old}_{s,m}$ can *only* be deleted if the summand on the right of the inequality in equation 5 equals 1—*i.e.*, if no predicates requiring $R^{old}_{s,m}$ are left to move to the new policy.

Equations 6 and 7 define $Z^{new}_{k,s}$ ($Z^{old}_{k,s}$) which gives the total number of new (old) rules that are added (deleted) to the switch s by the end of round k. Equation 8 tallies the rules that remain on each switch after each round and a lower bound on the overhead O is calculated relative to the number of rules in M^{max}_s (*i.e.*, whichever is larger among the old and the new policy).

The last group of equations simply define the types of the variables in the program.

Capping Overhead. The MIP in Figure 6 minimizes the overhead after fixing the total update time (*i.e.*, the number of rounds). It is also possible to fix the overhead allowed and minimize the total time required to perform the complete update, as in Figure 7. More specifically, given a threshold T (across all switches or a switch specific threshold T_s) on rule space, we try to complete the update as quickly as possible. This involves introducing a new binary variable I_k (one per round) as in equation 11, which represent whether

$$\min \sum_{k=1}^{K} I_k \text{ subject to} \quad (10)$$

$$I_k \geq 1 - \frac{\sum_{n=1}^{N} X_{n,k}}{N} \quad \forall k \in [K] \quad (11)$$

$$T_s \geq (M_s^{old} + Z_{k+1,s}^{new} - Z_{k,s}^{old}) \; \forall k \in [K-1], s \in [S] \quad (12)$$

Figure 7: Modified MIP with switch rule caps.

$$\max \sum_{n=1}^{N} \sum_{k=1}^{K} X_{n,k} V_n \text{ subject to} \quad (13)$$

$$T_s \geq (M_s^{old} + Z_{k+1,s}^{new} - Z_{k,s}^{old}) \; \forall k \in [K-1], s \in [S] \quad (14)$$

Figure 8: Modified MIP for traffic volume

a round k is an intermediate round where the update is still in progress (in which case $I_k = 1$) or if the update is already complete by round k (i.e, $X_{n,k} = 1, \forall n \in [N]$ in which case $I_k = 0$). The objective then is to minimize the number of such intermediate rounds as described in equation 10 (which replaces 1) subject to the additional constraints on I_k (equations 11) and rule space (equation 12).

Optimizing Traffic Volume Transfer. If information concerning flow volume is available, we can further adjust the MIP to update the parts of the policy handling the highest traffic volume as early as possible. Here, we assume each ingress predicate P_n is associated with a weight V_n that represents the estimated incoming traffic volume for that slice. Figure 8 illustrates the changes. In particular, the new objective function is represented by equation 13 and the additional rule space constraint 14 is added.

4. EVALUATION

We implemented the incremental updates algorithm in Python on the top of the popular NOX platform [1]. In this section, we present some preliminary results that show the scalability and flexibility of the update algorithm. All the experiments were done on three widely-used network topologies (using Mininet)—Fattree, Smallworld and Waxman, each with 24 switches and 576 hosts. Our experiments show the result of replacing one load-balancer (LB) policy with another. We chose to focus on this kind of policy as it is fairly challenging for analysis as it contains both complex forwarding and packet modification. To define an LB policy, a set of server replicas are allotted fixed nodes on the network. For every flow coming through an edge switch into the network, the policy assigns a random server replica, *modifies* the destination IP address of the flow to the assigned server IP address and then establishes the shortest path hop-by-hop from the edge switch to the server's location. The second LB policy is generated by bringing down a couple of servers and repeating the above procedure for a different random assignment of incoming flows to replicas. This resulted in around 100 OpenFlow rules per switch, each rule having an exact-match predicate on srcIP and dstIP with the corresponding actions including a forwarding action and an occasional header modification.

We implemented the reachability analysis in Python to obtain the weights used in the MIP; in all our experiments, it

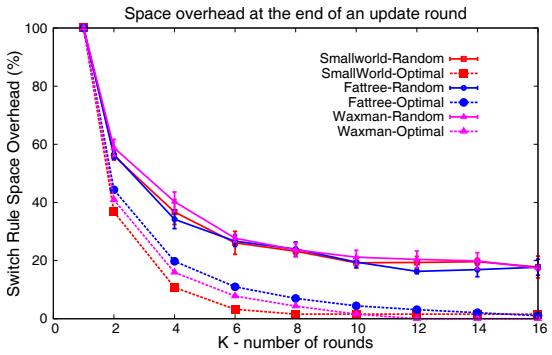

Figure 9: The rule space/time tradeoff

completed in approximately 2 minutes. Though this was not a particularly efficient implementation, recent research has shown that reachability analysis can be done in real time [4]. The MIP consisted of around 90,000 integer variables and was solved using the Gurobi optimizer [2]. This solver uses an iterative process to converge on an optimal solution. Consequently, as shown in Figure 10, one can trade computation time against the accuracy of the result, coming closer to the optimal solution as more time is spent. In our experiments, the Gurobi optimizer would return the optimal solution in a few seconds for most runs, but would occasionally run into hours in order to find the exact optimal solution. As shown in the figure, when the number of rounds increase, Gurobi takes more time preprocessing the MIP to prune the search space. However, after the preprocessing step, in most cases, we found the optimality gap drops to below 1% in about 5 seconds and 0.1% in at most 5 minutes, which was accurate enough for practical scenarios.

Figure 9 shows that incremental updates reduce rule-space requirements dramatically. The single point at the top left-hand side of the graph represents the rule-space overhead required for the consistent update algorithm formulated by Reitblatt *et al.* [8]: near 100% (*i.e.*, double the requirements of an individual policy). The graph shows 6 curves: 3 (upper) curves for *random* incremental updates (one curve for each of 3 different topologies) and 3 (lower) curves for *optimal* consistent updates. A *random* incremental update maps a random set of $\frac{N}{K}$ initial predicates to each of the K rounds and then does the reachability analysis in order to do a network-wide consistent update in each round. The *optimal* algorithm uses the MIP described in Figure 3. The upper 3 curves in Figure 9 demonstrate that even randomly-selected incremental updates are an excellent implementation strategy when compared with prior consistent update algorithms. They have the advantage that they are cheap to compute and yet reduce rule-space overhead considerably: Down to roughly 60% after just 2 rounds and close to 20% after 8-10 rounds. The overhead shown is the mean over 10 rounds with the standard deviation at a maximum of 4%. But if one has the time to solve the MIP, then one can achieve the best possible gains on the switch space overhead. Figure 9 shows that the space overhead decreases in proportion to the number of rounds used, touching almost 0% as K becomes large enough.

Figures 11 and 12 analyze the incremental update algorithm from the perspective of the extensions proposed to the MIP in Section 3. In these experiments, the switch rule-space overhead on any switch is capped at a maximum of

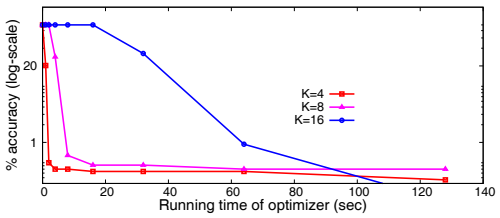

Figure 10: Gurobi accuracy Vs running time

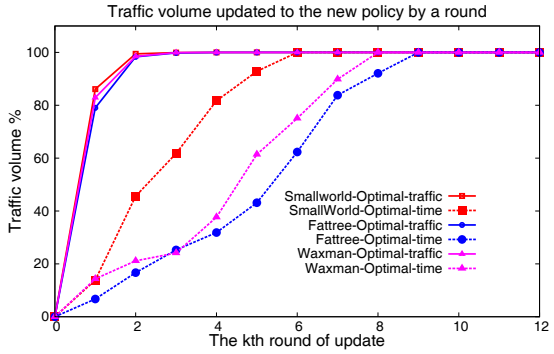

Figure 11: Optimizing for traffic volume

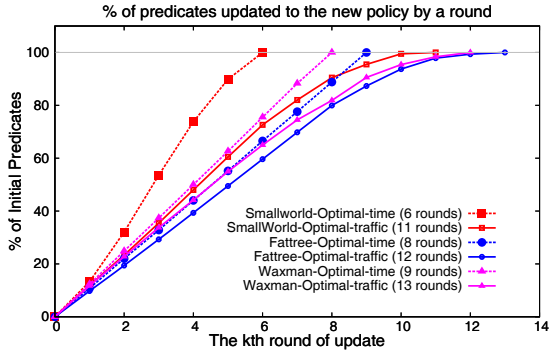

Figure 12: Optimizing for update time

5%. Given this threshold, Figure 11 asks how can one maximize the amount of traffic seeing the new policy as early as possible. This is done by assigning each initial predicate a weight which is a measure of the traffic volume handled by that predicate in an average time bin. The weights are drawn from a Zipf distribution in line with measurements reported in recent literature [9] on the skewed distribution of traffic volume over the prefixes/flows seen in ISPs, datacenters, etc. The figure shows the percentage of traffic covered by the updated policy by the end of each round of the incremental update process. As one can observe, almost 80% of the traffic volume shifts to the new policy by the end of the first round and 99% of the traffic shifts in just 3 rounds. But because of this, there is a tradeoff in terms of inefficient switch space usage which leads to a slight increase in the number of rounds needed to complete the update. Figure 12 shows the number of initial predicates that have been updated to the new policy by the end of a round. The figure also shows the number of rounds it takes to complete if the objective was to finish the update process as quickly as possible given the space threshold of 5%. One can observe that the number of rounds taken to complete increases as one shifts from the objective of minimizing time to maximizing traffic. Similarly if one aims to minimize the total update time (a function of K), then one has to tradeoff the volume of traffic seeing the new policy in the early rounds as one can see from Figure 11. Also, there is no obvious random predicate assignment algorithm that achieves these objectives while adhering to the space cap.

5. CONCLUSION

In this paper, we presented an algorithm for incremental consistent network update that navigates the trade-off between *update time* and *rule-space overhead*. It divides a global policy into a set of *slices* and by combining two important techniques—reachability analysis and an encoding as a mixed integer-linear program, shifts from old to new policy one slice at a time in an efficient manner. By increasing the number of slices, we substantially reduce the rule-space overhead. Moreover, when traffic volume follows Zipf's law, we also manage to shift most traffic to the new policy after just a few rounds of the algorithm.

Acknowledgments. The authors wish to thank the HotSDN reviewers and members of the Frenetic project for their feedback. We would also like to thank Mark Reitblatt and Nate Foster for sharing parts of their implementation of consistent updates. This work was supported in part by the NSF under the grant CNS 1111520; the ONR under award N00014-12-1-0757; and a Google Research Award.

6. REFERENCES

[1] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3), 2008.

[2] Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2012. `http://www.gurobi.com`.

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

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

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

[6] A. X. Liu, C. R. Meiners, and E. Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.*, 18(2):490–500, Apr. 2010.

[7] R. McGeer. A safe, efficient update protocol for OpenFlow networks. In *HotSDN*, Aug. 2012.

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

[9] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging Zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev.*, Jan. 2012.