NORTHWESTERN UNIVERSITY

Electrical Engineering and Computer Science

Compiling Minimum Incremental Update for Modular SDN Languages

March 26, 2014

X. Wen, C. Diao, X. Zhao, Y. Chen, L. Li, B. Yang, K. Bu

Technical Report NU-EECS-14-01

Abstract

Measurement results show that updating rules on switches poses major latency overhead during the course of the policy update. However, current SDN policy compilers do not handle policy updates well and generate large amount of redundant rule updates, most of which modify only the priority field. Our analysis shows that the lack of knowledge on the rule dependency and the consecutively distributed priority numbers are the fundamental problems behind the redundancy. In this paper, we tackle the problems through 1) a series of efficient algorithms that build rule dependency along with the compilation, and 2) an online optimization algorithm that maintains a scattered priority distribution with constant amortized cost. Our prototype evaluation demonstrates that our proposed patch eliminates nearly all the priority updates.

Compiling Minimum Incremental Update for Modular SDN Languages

Xitao Wen
Northwestern University

Chunxiao Diao
Northwestern University

Xun Zhao
Tsinghua University

Yan Chen
Northwestern University

Li Erran Li
Bell Labs, Alcatel-Lucent

Bo Yang, Kai Bu
Zhejiang University

ABSTRACT

Measurement results show that updating rules on switches poses major latency overhead during the course of the policy update. However, current SDN policy compilers do not handle policy updates well and generate large amount of redundant rule updates, most of which modify only the priority field. Our analysis shows that the lack of knowledge on the rule dependency and the consecutively distributed priority numbers are the fundamental problems behind the redundancy. In this paper, we tackle the problems through 1) a series of efficient algorithms that build rule dependency along with the compilation, and 2) an online optimization algorithm that maintains a scattered priority distribution with constant amortized cost. Our prototype evaluation demonstrates that our proposed patch eliminates nearly all the priority updates.

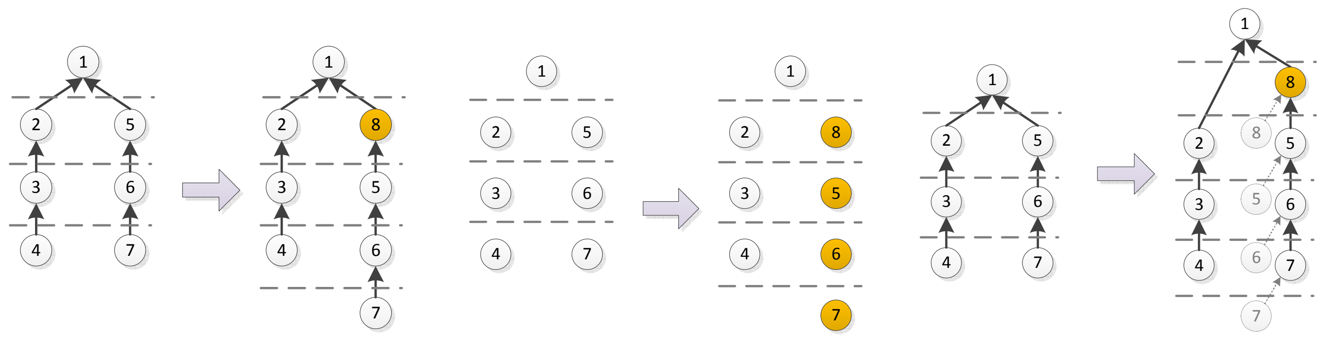
1. INTRODUCTION

Control plane modules are dynamic. The forwarding policies generated by the controller modules often have to dynamically react to network events with changes to the forwarding behaviors. Static forwarding policies that are too large to fit in a single flow table may also need dynamic swapping in reaction to the changing traffic patterns [4]. Although SDN allows a centralized approach to modify the forwarding policies installed in distributed switches, it does not avoid the significant latency overhead in altering the states of switches. According to the recent measurement results [3], state-of-the-art OpenFlow (OF) switches can only process tens to a few hundreds of flow-mod instructions per second, which implies that a full refresh of a 5K-length flow table could lead to tens of seconds of volatile inconsistent window on the data plane. Although recent proposals on consistent updates can eliminate the potential erroneous forwarding behaviors, they significantly increase the latency overhead by introducing multi-stage synchronization. Therefore, given a policy change, it is desirable to generate rule updates as compact as possible.

Policy composition of SDN policy languages makes the generation of compact updates challenging. In fact, parallel composition and sequential composition interleave the dependency relation among many rules and end up with one rule dependent on the policies from multiple modules. Our experiments show that a single-rule insert often results in modifying over half of the rules in a flow table with the straightforward update strategy on NetKAT [1]. The obscure dependency among rules forces the compiler to reassign a priority value to a rule even when its content aligns with an existing rule. The same problem applies to all current compilers that support policy composition. We observe the rule updates produced by several compilers, and identify two major types of rule updates: content update and priority update. Content updates, which involve the modification of the predicates or actions of the rules, are mostly direct results of the changing policy content. On the other hand, priority updates, which only modify the rule priorities, are mostly caused by poorly distributed priority levels and unnecessary priority shifts. Surprisingly, the priority updates often dominate the size of the total updates, implying that the poorly handled priority is the major blame for the inflated update size. In this paper, we focus on eliminating unnecessary priority updates.

Our observation reveals two fundamental problems that prevent the compilers from generating good priority updates.

1. **Missing rule dependency clue.** Although OF rules are associated with sequential priority values, the dependency relationship among rules actually forms a directed acyclic graph (DAG) [4]. Unlike the DAG representation that characterizes the minimum set of dependencies among rules, the sequential representation actually brings in plenty of non-existent dependencies and oftentimes makes avoidable priority updates necessary. Figure 1 shows an example where the dependency clue is essential to eliminate unnecessary priority updates.
2. **Consecutive rule priority.** All state-of-the-art SDN compilers generate rules with consecutive priority values for simplicity. However, consecutive priority values prevent an update generator from inserting rules between two adjacent priority levels without affecting either of them. In fact, OF specification allocates a 16-bit integer for flow priority that allows 65536 different priority levels, whereas the total number of priority levels is usually much smaller than 65536, implying the possibility to scatter priority values.



(a) An example of rule update. (b) Update without dependency clue. (c) Update with dependency clue.

Figure 1: Comparison of incremental update with vs. without dependency clue. In (b), due to the unawareness of the independency of the left and right rule chains, the update generator has to honor the priority levels set by the compiler and modify the priority of Rule 5, 6 and 7. While in (c), the optimal update requires only inserting Rule 8 in a new priority level.

In order to eliminate unnecessary priority updates, in this paper we propose 1) to guide update generation with rule dependency, and 2) to actively maintain the value gaps between priority levels.

The challenge of the first goal rests on how to obtain the dependency DAG. Intuitively, we can restore the dependency DAG directly from the output rules of the compiler with an existing algorithm [4]. However, we find such algorithm entails unacceptable $O(n^4)$ worst case computation complexity, where n is the size of the flow table. For this reason, we explore to build the dependency DAG gradually along the compilation process, which leads to a series of $O(n^2)$ algorithms that build dependency DAG during composition operations. With the dependency DAG, we can generate a provable minimum-size update with regard to continuous priority levels (i.e., new levels can be arbitrarily inserted between two levels).

When mapping to discrete priority values, priority updates are sometimes necessary to make room for new levels. Intuitively, scattered priority values reduce the chance of future priority shifts. Abstractly, to maintain scattered priorities requires an online optimization strategy that makes proactive priority updates to minimize the estimation of future priority shifts for an undetermined policy update sequence. In this paper, we propose *k-factor strategy* that maintains the lengths of all gaps between $[\frac{1}{k}, k]$ times the average gap length. K-factor strategy achieves k-factor gap distribution with $O(1)$ amortized cost of priority shifts.

To quantitatively estimate the benefit, we implement an initial prototype as an extension to NetKAT policy compiler, and evaluate it through benchmarks. Our experiments show that the dependency information and a scattered priority distribution enable the compiler to eliminate nearly all the priority updates, and reduce the update size by an order of magnitude.

2. BACKGROUND

Related Work.

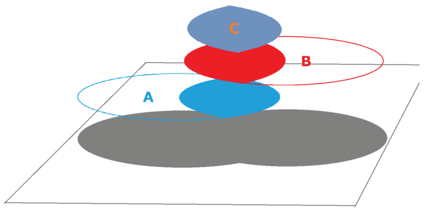


Figure 2: The intersection of Rule A and B is shadowed by Rule C. Thus A and B are independent, i.e. the vertical order does not change their match space.

Several SDN policy languages (e.g., Frenetic [2], NetCore [5] Maple [7] and NetKAT [1]) have been proposed in recent years. Generally, a policy language compiler takes high-level policy descriptions and generates flow tables that fulfill the semantics of the policies. Except Maple, most compilers do not provide any support for incremental policy updates. In practice, they simply compile the new policies and replace the entire flow table of each switch. A straightforward improvement can be made by updating only the rules whose content or priority changes. Yet for the aforementioned problems, a considerable number of unnecessary rule updates still have to be conducted. On the other hand, although Maple does not support policy composition, it introduces tree-style abstraction to support incremental flow table updates. However, Maple compiler still makes a large amount of priority updates due to the consecutive priority values.

Rule Dependency.

Intuitively, two rules are independent when their priority order does not change the header space classification. It is straightforward that two rules with disjoint predicates are independent. However, the opposite statement is not always true. For example, in Figure 2 the predicates of A and B intersect at a non-empty area, which is shadowed by the predicate of another rule C with higher priority. In this case, although $A \cap B \neq \varnothing$, they do not depend on each other in terms of representing the correct semantics. Therefore, we define two rules as *directly independent iff* the predicates of two rules are disjoint or their intersection is entirely shadowed by other rules they both depend on. This is different from the dependency defined in several other works [7, 4], which do not consider the shadowing area.

Further, we define the *indirect dependency*, or *dependency* for short, by taking a transitive closure on the direct dependency relations, *i.e.* rule A is indirectly dependent on rule B iff $\exists$ length-n sequence of rules Seq ($n \in \mathbb{N}^0$) that satisfies 1) A directly depends on Seq_1; 2) $\forall k < n, Seq_k$ directly depends on Seq_{k+1}; 3) Seq_n directly depends on B. Obviously, the dependency relation induces a strict partial order in a flow table, and therefore forms a DAG of rules. The dependency DAG reveals the inherent relationship among rules in a sense that it represents the minimum set of the priority constraints in order to keep the flow space classification semantics.

3. SOLUTION OVERVIEW

Ideally, a minimum policy update only involves the rules whose content has changed. Having the existing flow table and the updated one, it is easy to find the rules whose content (except priority level) holds. However, the priority values of these content-invariant rules may have changed in the new table. To avoid changing the content-invariant rules, we have to shift their priorities in the new table to the old values. Meanwhile, such shifts may require further changes on the rules that are dependent on the content-invariant rules, or the rules those content-invariant rules depend on. Therefore, the minimum set of rule dependency of the new table is the key in generating of the minimum policy update. Such logic leads us to the design of the update generation framework as shown in Figure 3.

Preserving Dependency in Compilation.

In our framework, the policy compiler outputs the rule dependency DAG to describe the dependency among rules. Since the current compilers typically maintain the dependency via priority value, it is feasible to restore the dependency DAG from the flow table through calculation. However, due to the high computation complexity of the restoration, we opt for the alternative approach to track the dependency during compilation (§4).

Comparing Updated Flow Table with Existing One.

To find the minimum rule update, the comparer identify all the content-invariant rules by comparing the updated flow table with the existing one. For those content-invariant rules, we will try the best to reuse their current priority values unless there is a new priority level inserted into two priority levels that are currently mapped to two consecutive priority values. Other newly arrived rules are tagged on the dependency DAG to feed the prioritizer. At the end, those newly arrived rules are realized either by modifying retired rules or inserting new ones.

Assigning Priority Values to Priority Levels.

Initially, the priority levels should be evenly scattered on the priority value space. Upon each update, the prioritizer assigns priority values to priority levels based on their old values and the new dependency relation. The prioritizer maintains the distribution of the priority gaps with certain online strategy.

4. PRESERVING RULE DEPENDENCY

There are two ways to obtain rule dependency: to restore from flow table after compilation, or to gradually construct during the compilation, especially the module composition. In our analysis, we find the dependency construction algorithm proposed in CacheFlow [4] has time complexity of $O(n^3) \sim O(n^4)$ with regard to the length of the flow table. In this section, we show the complexity analysis of the dependency construction algorithm. Meanwhile, we explore the latter approach and find a a series of $O(n^2)$ algorithms to build the dependency DAG during compilation. We describe these algorithms on the abstractions of NetKAT local compiler. The algorithms can be easily adapted to other compilers that conduct similar policy composition.

4.1 Constructing Dependency DAG from Flow Table

We modify the dependency construction algorithm of CacheFlow to reflect our slightly different definition of the rule dependency as depicted in Algorithm 1. We then analyze the runtime complexity of the algorithm.

```
input  : Flow table $T$
output: Dependency DAG $G$
foreach rule $R$ in $T$ in descending priority order do
    $C = R.match$;
    foreach $R_i$ in $T$ with $R_i < R$ in descending priority order do
        if $C \cap R_i.match \neq \emptyset$ then
            $G = G \cup (R, R_i)$;
            $C = C - R_i.match$;
            $R_i.match = R_i.match - R.match$;
        end
    end
end
```

Algorithm 1: Constructing the dependency DAG.

THEOREM 1. *Given the set expression representation of flow matches, the dependency construction algorithm has a time complexity of $O(n^4)$.*

PROOF. The time complexity of the algorithm is determined by the set operations, including set intersection at Line 4 and set difference at Line 6 and 7. The time complexity of set operations depends on the data structure of the match set, which is typically represented by the set expressions of unit predicates.

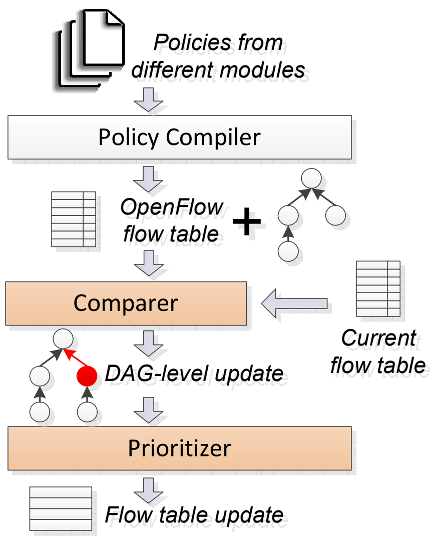


Figure 3: Overview of the solution.

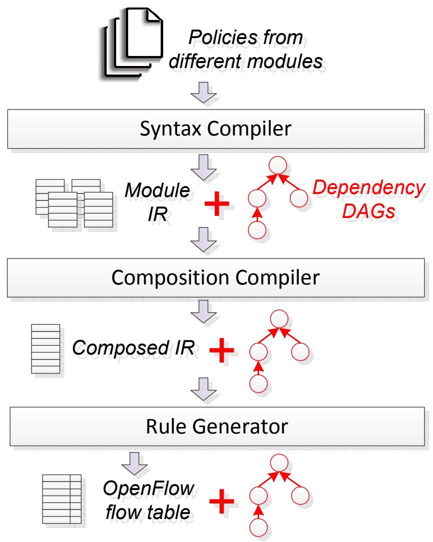
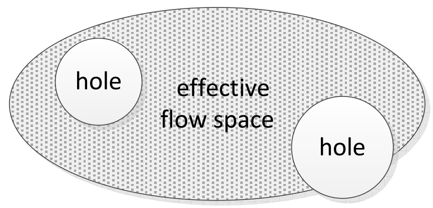
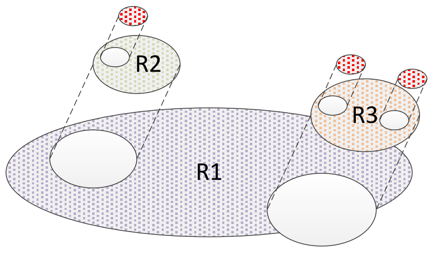


Figure 4: SDN policy compilation and our proposed patch.



(a) Match space of an ONF atom.



(b) Hierarchical dependency of ONF atoms.

Figure 5: Conceptual structure of ONF atoms.

Now we formally define the set expression representation of matches. We denote unit predicates as U_i, U_j, which represent any header sets that can be characterized with a single ternary header pattern (a string of 0, 1 or wildcard). Since unit predicates do not form a closure with regard to set operations, the results of set operations may be represented as expressions, such as $E_m = U_i \cup (U_j \cap U_k)$ and $E_n = U_i - U_j = U_i \cap \overline{U_j}$. Further, in order to determine whether a predicate expression is an empty set (Line 4 of Algorithm 1), the expression has to be reduced to disjunctive normal form (DNF) or conjunctive normal form (CNF). We define the length of an expression as one plus the minimum number of union operations in any equivalent form of the expression. For example, the length of E_m and E_n is 3 and 2 respectively.

With the above the set expression representation, it is easy to see the time complexity of the union operation is linear to the product of the length of both operands. We can show that in worst cases the length of result expressions of set difference is linear to the sum of the length of both operands, namely $O(D) = O(N_1 + N_2)$. In fact,

$$\begin{aligned}
&E_m - E_n\\
=&E_m \cap \overline{E_n}\\
=&(\bigcup_i \bigcap_j U_{mij}) \cap (\overline{\bigcap_i \bigcup_j U_{nij}}) \text{ (Definition of DNF and CNF)}\\
=&(\bigcup_i \bigcap_j U_{mij}) \cap (\bigcup_i \bigcap_j \overline{U_{nij}}) \text{ (De Morgan's Law)}\\
=&\bigcup_{i,k} (\bigcap_j U_{mij} \cap \bigcap_j \overline{U_{nkj}}) \text{ (Distributive Property)}
\end{aligned}$$

The main structure of Algorithm 1 is a nested loop. Originally, the match of each rule (i.e., $R_i.match$) has a constant length. However, as the outer loop proceeds, the length of R_i grows linearly with the iteration index (Line 7). Thus the initial length of the cumulative variable C of the inner loop is linear to the iteration index of the outer loop. Consider the runtime of Line 4,

$$\begin{aligned}
&O(T)\\
=&O(\sum_{i=1}^{n} 1 * i + \sum_{i=2}^{n} 2 * i + ... + \sum_{i=n}^{n} n * i)\\
=&O(\sum_{o=1}^{n} \sum_{i=o}^{n} o * i)\\
=&O(\sum_{o=1}^{n} o * (n-o) * (n+o))\\
=&O(n^4)
\end{aligned}$$

□

4.2 SDN Compiler Background

The compilation of high-level SDN policy languages typically contains three major tasks, as depicted in Figure 4. First, the *syntax compiler* transforms a module's policy from the high-level language to an intermediate representation (IR) of rule tables. Second, the *composition compiler* combines the potentially conflicting rule tables from different modules into a consistent rule table, according to the composition relationship among modules. Third, the *rule generator* translates the IR of rule tables into OpenFlow-compatible flow table.

To preserve the dependency DAG, we propose to explicitly preserve dependency information during all three tasks.

First, we need the syntax compiler to output the dependency DAG, since it is a natural side product of the syntax compilation. Second, we need the composition compiler to maintain the dependency DAG during composition. Third, since the rule IR may not have a one-to-one mapping with the final OF rules, the rule generator also needs to maintain the dependency DAG as it adds or merges rules.

NetKAT uses OpenFlow Normal Form (ONF), a subset of NetKAT language, as the intermediate representation towards OpenFlow flow table. An ONF flow table is comprised of abstract ONF rules, or atoms. Structurally, ONF forms a degenerate if-else binary tree representing the abstract atom sequence. Conceptually, the predicate of each abstract atom can be seen as a flow space filter with an outline match and possibly some holes representing the effect of the atoms it depends on, as depicted in Figure 5.

The compilation of NetKAT can be mapped to the compilation framework in Figure 3. NetKAT compiler first transforms the predicates and actions into if-else binary trees, corresponding to the syntax compilation step. Then, the compiler eliminates the composition operators through ONF tree transformation, corresponding to the composition compilation step. Finally, the rule generator transforms the abstract atoms into concrete flow rules, corresponding to the rule generation step.

As depicted in Figure 3, our goal is to explicitly build the dependency DAGs along each step of the the compilation. Particularly, during the compilation each ONF fragment should be associated with a dependency DAG, whose vertices represent ONF atoms and edges represent the dependency between atoms.

4.3 Initiating Dependency DAG in Syntax Compilation

After syntax compilation, NetKAT policies are essentially ONF fragments connected by composition operators. Particularly, at this stage each ONF fragment contains only one ONF atom. Therefore, the dependency DAGs associated with all of the ONF fragments are the same single-node graphs.

4.4 Preserving Dependency DAG in Composition

At this stage, NetKAT compiler recursively eliminates the composition operators and gradually combines the ONF fragments into a single ONF flow table. In each recursion, the compiler combines two ONF fragments connected by a composition operator into a single ONF fragment. Since each composition operator also combines the dependency DAGs of two ONF fragments, the goal of our extension is to combine and maintain the dependency DAG during NetKAT policy composition.

Parallel Composition.

Generally, the result of parallel composition contains three parts of policies, i.e., $S_1 = P_1 - P_2$, $S_2 = P_2 - P_1$ and $S_3 = P_1 \cap P_2$, as shown in Figure 6. Since dependency essentially can be seen as the holes in the match space, intuitively each of the three parts inherits a subset of the dependency (or holes) from the operands. Specifically, the parallel composition of two ONF fragments is calculated by taking cross-product of predicates and actions. Thus, each rule in S_1 (or S_2) originates from a rule in P_1 (or P_2). Because the match of the new rule is a subset of that of the original rule, it inherits a subset of the dependency from the original rule. Similarly, the rules in S_3 may inherit dependency from both operands. *Therefore, a superset of the result dependency DAG can be obtained by taking the cross-product of the operands' dependency DAGs.*

Then, the edges of the result DAG have to be validated. We can simply take the intersection of the two rules on both ends of an edge. The dependency is still valid if and only if the intersection is not empty.

At last, one special treatment has to be made on the holes in S_3. Since these holes are compiled to new rules, we need to add dependency between them and S_3.

Sequential Composition.

The result of sequential composition only contains one part of rules, i.e., $P_1 \cap P_2'$ in Figure 7. In fact, the sequence of two ONF fragments P_1 and P_2 can be seen as the union of P_1 and P_2', which is the projection of P_2 prior the actions of P_1. Since the actions preserve the dependency relations, the result ONF fragment inherits a subset of the dependency from both operands. In other words, considering a result rule R that originated from $R_1 \in P_1$ and $R_2 \in P_2$, R may be dependent on every rule that originates from R_1 and R_2. Like in parallel composition, all the dependency relations have to be validated by checking the overlapping of the two relevant rules.

Priority Composition.

To demonstrate the flexibility of the dependency DAG, we also devise a new binary composition operator, priority composition. The semantics of the priority composition is like setting different priority levels for two operands, as depicted in Figure 8. In other words, the second operand only takes effect on the flows that do not match the first operand. Generally, the priority union of two ONFs is calculated by taking a literal union of the operands' ONF rules with proper priority configuration.

The compilation of priority composition is made possible by the flexibility of dependency DAG. As the example in Figure 8, we just need to set the rules from P_1 to be inferior to the rules from P_2. Then, similar with other compositions, we validate the added dependency by checking the overlapping. Also, the holes have to be treated specially.

4.5 Maintaining Dependency DAG in Rule Generation

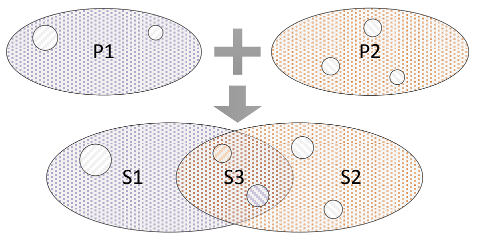


Figure 6: Parallel composition's effect on dependency DAG.

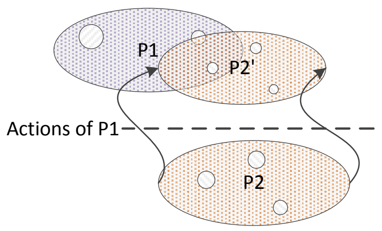


Figure 7: Sequential composition's effect on dependency DAG.

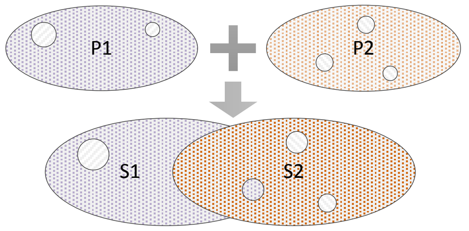


Figure 8: Priority composition's effect on dependency DAG.

In the rule generation stage, the rule generator splits an ONF atom into one base OF rule plus multiple shadow OF rules (i.e., the holes). The effect to the dependency DAG is treated in Algorithm 2.

```
input  : Dependency DAG $G = (V, E)$, Mappings from
         DAG vertices to OF base rule and shadow
         rules $M : v \to \{b, S\}$
output: New dependency DAG $G' = (V', E')$
$G' = (\emptyset, \emptyset)$;
foreach Mapping $v \to \{b, S\}$ do
    $V' = V' + \{v_b, v_s^1, ..., v_s^{|S|}\}$;
    foreach Edge $e = (u, v) \in E, u \neq v$ do
        $E' = E' + (u, v_b)$;
    end
    foreach Edge $e = (v, u) \in E, u \neq v$ do
        foreach $i \in [1, |S|]$ do
            if $v_s^i.match \cap u.match \neq \emptyset$ then
                $E' = E' + (v_s^i, u)$;
            end
        end
    end
    foreach $i \in [1, |S|]$ do
        $E' = E' + (v_b, v_s^i)$;
    end
end
```

Algorithm 2: Maintaining Dependency DAG in Rule Generation

5. MAINTAINING SCATTERED PRIORITY VALUES

Assigning priority values is an online strategy: upon each update, the prioritizer assigns the priority values to all new rules and possibly some old rules without the knowledge of the future updates. Intuitively, a more evenly scattered distribution of priority values reduces the chance of future priority updates with the cost of proactive priority updates. In this section, we describe the k-factor strategy, which actively maintains all gap lengths within the range of $[\frac{1}{k}, k]$ times the average gap length l_a, where k is a configurable parameter within the range $[1, +\infty)$. The k-factor strategy costs amortized $O(1)$ maintenance updates to achieve its gap distribution.

Without loss of generality, we assume the update is an insert or delete of a single rule. K-factor strategy works as follows. It first assigns all content-invariant rules with old priority values.

- If the update is a single-rule insert, the priority level of the new rule must be located between two existing priority levels on the priority DAG. Thus, the prioritizer allocates a new priority value between the two levels, and the gap is halved. If the lengths of the new gaps are less than the lower bound limit l_a/k, the prioritizer must shift the neighboring levels to meet the limit. Particularly, denoting the new level as mth level, the prioritizer determines the least number of neighboring levels n that satisfies $\lfloor(\Sigma_{i=m}^{m+n} l_i)/n\rfloor \geq l_a/k$ or $\lfloor(\Sigma_{i=m-n}^{m} l_i)/n\rfloor \geq l_a/k$. Finally, the prioritizer shifts the priority values of the n neighboring levels to equally partition the gaps.
- The process is similar for a rule delete except that the prioritizer now tests the upper bound limit kl_a instead of the lower bound limit l_a/k.

The parameter k balances the evenness of the gap distribution and the maintenance cost. At one end of the spectrum, when k equals 1, the prioritizer always maintains a uniform priority distribution, which costs more priority shifts for maintenance. At the other end, if k is large enough, the prioritizer only make priority shifts when no middle value is available for assignment.

6. EVALUATION

In this section, we evaluate how much size of the rule updates our proposed techniques can reduce.

Methodology.

We implement a prototype of the update generation framework based on NetKAT policy compiler. Our extension comprises two components: the DAG generator and the k-factor prioritizer. The DAG generator constructs the dependency graphs from prioritized OF rules with an algorithm adapted from CacheFlow. The k-factor prioritizer compares the new rules with the existing ones and assigns priority value to the incremental rules with the k-factor strategy. We slightly modify NetKAT to allow the bit-wise IP prefix matches in our policy benchmark. We set parameter k to 2.

	Total	2%			4%			6%		
	#Rules	Naive	DAG	Optimal	Naive	DAG	Optimal	Naive	DAG	Optimal
fw1	898	592.40	52.03	37.06	589.39	92.10	66.03	493.81	113.25	79.23
fw2	376	215.06	19.94	14.78	240.21	31.81	23.36	251.64	44.37	32.31
acl1	121	56.50	3.19	2.61	71.58	7.18	5.46	80.37	11.71	8.77
acl2	594	91.17	8.00	6.30	420.76	78.08	55.58	346.50	84.45	60.84
ipc1	303	119.79	11.69	9.05	130.38	18.67	13.94	173.18	32.69	23.96
ipc2	243	100.00	6.98	5.27	105.47	12.30	9.27	173.67	28.99	20.76

Table 1: Number of rules and mean size of rule updates of 100 successive policy updates. Correspondingly 2, 4 or 6 percent of the filters are replaced in each round of policy update.

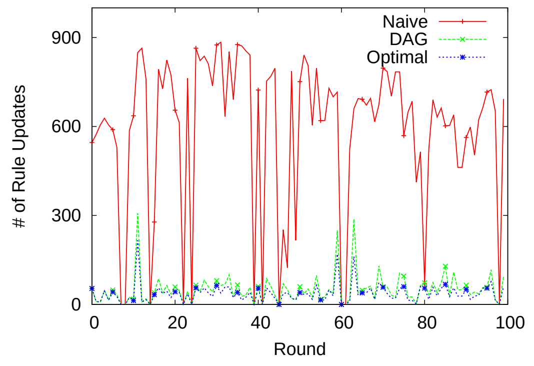


Figure 9: The update sizes of 100 successive policy updates on trace fw1. Two percent of the filters are replaced in each update.

We build the base policies from filter sets generated by ClassBench [6]. We build the initial policy by randomly selecting 100 filters from the filter set, and then generate policy updates by randomly replacing a portion (2%, 4% or 6%) of filters in the previous policy. For each configuration, we evaluate 100 successive rounds of policy updates.

We compare the update size generated by our framework (denoted as DAG) with 1) a naive strategy that updates the diff of flow tables, and 2) the optimal updates that contains no priority updates but only content updates.

Results.

Table 1 shows the mean size of the 100 rounds of updates for each configuration. And Figure 9 shows all the size comparison of all 100 rounds of updates at the configuration [fw1, 2%], which represents a typical case in our experiments. We observe that the update size of the DAG cases is close to the optimal cases and is usually one order of magnitude smaller than that of the naive cases.

7. CONCLUSION AND FUTURE WORK

In this paper, we propose an update generation framework that utilizes the rule dependency information to minimize the number of rules to be modified in a policy update. We present the key techniques in the framework including how to obtain dependency information and how to maintain the scattered distribution of priority values. We further evaluate the benefit of our framework through benchmarks.

For future work, we would like to implement the incremental dependency construction on NetKAT policy compiler and evaluate the runtime performance gain. We will also explore other priority assignment strategies, especially the one that makes constant priority shifts in worst cases.

8. REFERENCES

[1] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. NetKAT: Semantic Foundations for Networks. In *Proceedings of POPL '14* (2014), ACM, pp. 113–126.

[2] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. Frenetic: A network programming language. In *ACM SIGPLAN* (2011), vol. 46.

[3] Huang, D. Y., Yocum, K., and Snoeren, A. C. High-fidelity Switch Models for Software-defined Network Emulation. In *Proceedings of HotSDN '13* (New York, NY, USA, 2013), ACM, pp. 43–48.

[4] Katta, N., Rexford, J., and Walker, D. Infinite cacheflow in software-defined networks. Tech. rep., TR-966-13, Department of Computer Science, Princeton University, 2013.

[5] Monsanto, C., Foster, N., Harrison, R., and Walker, D. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 217–230.

[6] Taylor, D. E., and Turner, J. S. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw. 15*, 3 (June 2007), 499–511.

[7] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. Maple: simplifying SDN programming using algorithmic policies. In *Proceedings of SIGCOMM '13* (2013), ACM, pp. 87–98.