RuleTris: Minimizing Rule Update Latency for TCAM-based SDN Switches

Xitao Wen*, Bo Yang†, Yan Chen*, Li Erran Li‡, Kai Bu†, Peng Zheng§, Yang Yang*, Chengchen Hu§
*Northwestern University, †Zhejiang University, ‡Fudan University, §Xi'an Jiaotong University

***Abstract*—Software-defined network (SDN) is deemed to enable more dynamic management of data center networks that promptly respond to network events with changes in network policies. Although the SDN controller architecture is increasingly optimized for swift policy updates, the data plane, especially the prevailing TCAM-based flow tables on physical SDN switches, remains unoptimized for fast rule updates, and is gradually becoming the primary bottleneck along the policy update pipeline.**

In this paper, we present *RuleTris*, the first SDN update optimization framework that minimizes rule update latency for TCAM-based switches. *RuleTris* employs the dependency graph (DAG) as the key abstraction to minimize the update latency. *RuleTris* efficiently obtains the DAGs with novel dependency preserving algorithms that incrementally build rule dependency along with the compilation process. Then, in the guidance of the DAG, *RuleTris* optimizes the rule updates in TCAM to avoid unnecessary entry moves, which are the main cause of TCAM update inefficiency. We prove that *RuleTris* generates TCAM updates with the minimum number of TCAM entry moves. In evaluation, *RuleTris* achieves a median of $<$12ms and 90-percentile of $<$15ms the end-to-end per-rule update latency on our hardware prototype, outperforming the state-of-the-art composition compiler CoVisor by $\sim$20 times.

I. Introduction

One of the key capacities promised by software-defined network (SDN) is the ability to dynamically change the network states in response to the global view. Based on how fast network states can respond to network events, lots of new network applications can become practical. For example, carrier network has a strict 50ms requirement for failure recovery [1], entailing a 10ms to 25ms delay budget for implementing the rerouting rules. Traffic engineering in data centers has a delay budget as short as 100ms for the entire control loop [2], leaving less than 20ms delay budget for implementing flow rules. Advanced malware quarantine [3] in enterprise networks has an even stricter delay budget since the threat detection is done at near line-rate and the quarantine decisions need to take effect as fast as possible.

The recent advances on SDN controller architecture greatly shorten the processing latency of the control plane, which leaves the rule installation latency the primary bottleneck for the SDN control loop. Specifically, the recent measurement [4] exhibits a rule installation delay ranging from 33ms to 400ms with a moderate to high flow table utilization on three commercial OpenFlow switches using ternary content-addressable memory (TCAM), which is the mainstream hardware to implement OpenFlow compatible flow tables [1]. In addition, the measurement also finds that the switches can *"periodically or randomly stop processing control plane commands for up to 400ms"*, which further exacerbates the rule installation latency.

Although some existing works optimize the policy updates at different stages of the pipeline, their improvements are limited. Dionysus [5], for example, significantly reduces multi-switch policy update latency caused by suboptimal scheduling. CoVisor [6] and our previous short paper [7] minimize the number of rule updates sent to switches through eliminating redundant updates. However, since both approaches do not change the update mechanism on physical switches, they all suffer from the aforementioned per-rule update bottleneck. Existing TCAM update optimization techniques, on the other hand, are either dependent on specialized multi-stage SRAM/TCAM structure [8], [9], [10] or only applicable to single-field longest prefix matching [11].

In this paper, we aim to tackle the update latency bottleneck on the TCAM-based SDN switches. Our measurement and analysis exhibit that the TCAM update latency is *the single dominant factor* of the rule update latency. Interestingly, although a single entry move in TCAM usually has a constant sub-millisecond delay, we observe that an OpenFlow rule update sometimes triggers hundreds to thousands of unnecessary entry moves in TCAM to maintain rule dependency due to its unawareness of the minimum dependency information.

We present *RuleTris*, the first SDN update optimization framework that minimizes rule update latency in TCAM. Our study reveals that the minimum *dependency graph* (DAG) [10], [12], [13] is the key information towards optimal rule updates. *RuleTris* is comprised of a front-end and a back-end as depicted in Figure 1. The *RuleTris* front-end is a generic policy compiler that *produces DAGs* while composing multiple flow tables. The DAG produced by the front-end along with the flow table is then passed to the back-end for update optimization. The *RuleTris* back-end is a set of hardware-specific optimizers that map the DAG into a sequence of TCAM entry moves. The optimizers minimize the flow table size and the number of entry moves by exploiting the minimum dependency information.

To realize such an optimization framework, the primary challenge is to generate DAG efficiently. In fact, the existing DAG extraction algorithm is prohibitively time consuming for our target latency [13]. To this end, we embrace the *policy composition paradigm* [14]. Our previous short paper proposes to preserve rule dependency within NetKAT policy compiler [15] to reduce computation. Extending it for *generic policy compilation* is quite non-trivial since a common flow table abstraction needs to be employed in the dependency reservation algorithms. Furthermore, to minimize

[1]Our survey indicates that at least 32 out of all 48 series of OpenFlow supported switches from 13 major vendors use TCAM to implement OpenFlow compatible flow tables.

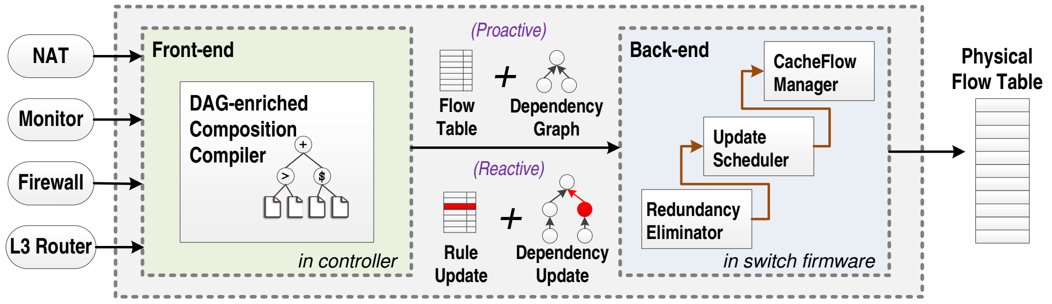


Fig. 1: Overview of *RuleTris* optimization framework.

the compilation overhead, the DAG needs to be *compiled incrementally* as policies evolve over time. On the back-end, an optimal while efficient scheduling algorithm is also needed to map the incremental graph changes into minimum TCAM entry moves.

RuleTris solves these challenging problems with the following contributions.

1) We develop general dependency preserving algorithms that preserve DAG along with flow table composition. The algorithms achieve efficiency by exploiting the dependency implications of composition operators. The algorithms are generic to SDN policy languages that employ policy compositions (sequential, parallel and priority), and are guaranteed to produce the minimum DAG.
2) We further speed up the compilation by incrementally compiling flow table changes. We employ incremental compilation techniques and develop algorithms to handle incremental DAG compositions.
3) We develop efficient back-end scheduling algorithms to map incremental DAG changes to rule updates in TCAM. Our back-end components optimize the rule updates to achieve *provably minimum* entry moves in TCAM, eliminate redundant rules and provide support for efficient rule caching hierarchy to scale up the size of flow tables.

RuleTris can be deployed in a variety of settings. It can be embedded to a policy compiler, so that minimum updates can be generated even for these incremental-agnostic SDN applications that populate non-minimum rule updates. It can also be built as extensions of SDN controllers or controller hypervisors, so that the policy composition of multiple SDN applications or controllers is updated with minimum number of operations.

We fully implement *RuleTris* front-end as a standalone composition compiler, and the back-end in the firmware of the data-plane programmable hardware-based ONetSwitch [16], [17]. Through hardware evaluation, we demonstrate that *RuleTris* achieves a median of $<$12ms and 90-percentile of $<$15ms the per-rule update latency, outperforming the state-of-the-art composition compiler CoVisor deployed on the same hardware switch by $\sim$20x. Our large scale emulation indicates even greater speedup on larger TCAM size.

Roadmap. We give background in §II, followed by an overview in §III. We describe the front-end design in §IV and back-end design in §V. We present our implementation in §VI, evaluation in §VII, provide discussions on future topics in §VIII. We review related work in §IX and conclude in §X.

II. Background

a) Rule Updates on Physical Switches: TCAM is the mainstream hardware to implement flow tables in hardware SDN switches. Although TCAM offers incomparable lookup performance, current commercial TCAM solutions are slow on rule update. Measurement studies show that a single rule update can bring tens to hundreds of milliseconds of data plane disruption on state-of-the-art switches [4], [18], since typically conducting updates requires locking TCAM from accepting data plane lookup requests.

Maintaining rule dependency is the main reason to blame for the slow updates of TCAM. In fact, one rule update from the controller can often result in massive TCAM entry moves. This is because TCAM implements rule dependency using the relative physical location [11], [19], i.e., a rule located at a higher physical address has a higher matching priority. Although other dependency encoding schemes have been proposed, the physical location encoding is still the mainstream implementation today[20]. Upon the arrival of a new rule, the switch firmware [2] may have to move many existing entries in order to keep the correct rule dependency. Furthermore, since multiple TCAM entry updates cannot be conducted in parallel, the massive TCAM moves eventually lead to significant rule update latency. The approach *RuleTris* takes to minimize rule update latency is to eliminate unnecessary TCAM entry moves through maintaining a minimum DAG.

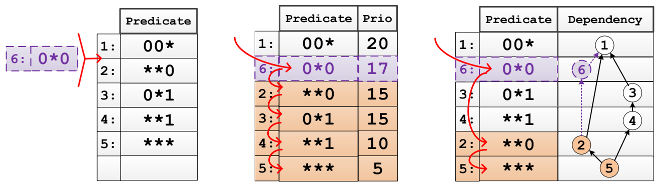
b) Rule Dependency: The predicate of a rule specifies the flow space the rule should match. When two rules have an overlapping predicate, the matching ambiguity needs to be resolved by specifying a matching order. In the context of a flow table, we define the rule dependency as the relation between a pair of rules if their matching order changes the actual rule matching semantics. Without loss of generality, we say Rule A *is dependent on* Rule B if Rule B should be matched first.

Obviously, the dependency relations form a directed acyclic graph, or DAG [10], [12]. The minimum DAG reveals the inherent relationship among rules in a sense that it represents the minimum set of the matching order constraints in order to keep the correct classification semantics of flow space. In this paper, we use the term *DAG* to refer specifically to the minimum dependency graph of a flow table.

In fact, assigning rules with integer priority values is the way OpenFlow employs to unambiguously represent rule dependency. However, rule priority does not directly induce a set of minimum dependency relations in a sense that two rules with different priority values are not necessarily dependent.

c) Modular Composition: Modular composition was widely used in network programming languages and hypervisors to provide transparent composition and collaboration of control plane applications [6], [14], [15], [21]. In this

[2] Also refered as TCAM ASIC driver in the switch OS.



(a) An example of rule update. (b) Update with priority. (c) Update with DAG.

Fig. 2: An example rule insert in a TCAM table. The original TCAM table has five entries (Rule 1-5) and one empty slot in the end. Rule 6 needs to be inserted between Rule 1 and Rule 2. In Fig(a), the firmware schedules the insertion plan according to the dependencies implied by the priority values, therefore Rule 2 through Rule 5 are moved in order to preserve their relative positions. In Fig(b), however, the DAG indicates the newly inserted Rule 6 has no dependency with Rule 3 and Rule 4, therefore only Rule 2 and Rule 5 needs to be moved.

paper, we compose applications with three composition operators: *parallel operator*, *sequential operator* and *priority operator*. Parallel operator (+) creates the illusion that multiple applications to independently process the same traffic. Sequential operator (>) allows an application to process the traffic before another. Priority operator ($) gives an application the priority to act on some traffic while yielding the control of the rest to other applications.

A composition compiler is typically used to compile the composition of applications into a semantically equivalent flow table to install on the physical switches. Since applications can act on different header fields, the result flow table usually contains many rules that overlap with each other. All existing composition compilers use priorities to keep the dependency.

III. RuleTris Overview

In this section, we first motivate the necessity of the DAG with an example in §III-A. We then depict *RuleTris* optimization framework in §III-B, followed by the optimality claims in §III-C.

A. Benefits of DAG

Generally, optimally updating TCAM in physical switches requires a minimum DAG. In implementing a rule update in the TCAM, integer priority values provide the complete dependency information and thus can be used to generate semantically correct update schedule. For example, in Figure 2, Rule 6 is to be added to the flow table. As shown in Figure 2(b), according to the relative priorities, Rule 6 should be placed at a slot with a higher physical address than Rule 2 through Rule 5 and a lower address than Rule 1. Since the only available slot is at the bottom, each of Rule 2 through Rule 5 has to move one slot down to make room for Rule 6.

However, priority values do not guarantee optimality in rule updates. In fact, the integer priority representation implies that all rule pairs with different priority values have dependency, which introduces a huge amount of non-existing dependency constraints. During the rule update, the redundant dependencies lead to unnecessary TCAM entry moves.

Instead, the DAG represents a minimum set of dependency constraints and guarantees to produce the optimal update schedule (we will show the optimality in §III-C). For example, Figure 2(c) shows the optimal update schedule guided by the DAG. Since Rule 6 and Rule 2 has no overlapping flow space with Rule 3 and Rule 4, the optimal update schedule only needs to make two extra entry moves instead of four.

The above example shows the benefit of the DAG in scheduling rule updates. In fact, maintaining the DAG provides a series of other benefits. For example, the DAG makes it straightforward to generate a flow table without rules that are entirely obscured by higher priority rules. By scanning the flow-table in the topological order of the DAG, we can easily eliminate the redundant rules that will never be matched or do not alter the data plane behavior. Also, DAG enables an efficient way to support arbitrarily large flow tables through rule caching [13].

B. End-to-End Optimization Framework

The above example shows the importance of the DAG, and leads us to the design of *RuleTris* optimization framework as in Figure 1. *RuleTris* optimization framework is comprised of the front-end composition compiler and the back-end optimizers.

a) Front-end: *RuleTris* allows administrators to compose multiple controller applications or controllers through composition operators. Such capacity is provided by a general-purpose composition compiler that makes up the *RuleTris* front-end. *RuleTris* composition compiler interfaces with applications or controllers, accepting their proactive or reactive modification of the network policies. Similar with other composition compilers, *RuleTris* composition compiler is configured by the administrator to compose the application policies into a single policy implementation for physical network devices.

Except for the compiled flow tables, *RuleTris* further generates the DAGs to resolve the matching ambiguity, which replaces the integer priority values used in other composition compilers. Upon the arrival of proactive network policy installation, *RuleTris* compiles the policies in batch, and supplies the back-end with a fresh flow table with the entire DAG. Upon the arrival of reactive policy updates, *RuleTris* compiles the policy updates in an incremental manner, and supplies the back-end with incremental rule inserts, deletes and modifications together with the updates to the DAG.

RuleTris does not require applications/guest controllers to be dependency-aware. If an application populates prioritized flow tables, *RuleTris* can extract the DAGs from the prioritized flow tables.

b) Back-end: The *RuleTris* back-end optimizers exploit the benefits of the DAG and optimize the actual rule installation/update process in the physical switches. For now, *RuleTris* provides three back-end optimizers. The *update scheduler* conducts hardware-specific optimization with DAG, and generates provably minimum-size update schedule to implement rule updates in TCAM tables. The *redundancy eliminator* removes two types of redundant rules. The *CacheFlow manager* manages multi-level rule cache and conducts rule eviction guided by the DAG [13]. *RuleTris* back-end directly generates sequence of TCAM entry moves.

c) Front-end/back-end communication: In this paper, we assume *RuleTris* back-end is co-located with physical switches. The front-end to back-end communication is carried through the control channel, e.g., OpenFlow. *RuleTris* extends OpenFlow protocol with DAG extension using the experimenter message, so as to allow protocol message to carry DAG or DAG update. Alternatively, *RuleTris* back-end can also be co-located with the front-end. In this way, no DAG extension is necessary but the control channel needs to be extended to expose the TCAM internal layout.

C. Optimality Guarantees

RuleTris provides several optimality guarantees as follows. We show how the optimality is achieved in Section V.

Claim 1. *With DAG, the back-end can generate the minimum number of entry moves that correctly implements a specific rule update in a TCAM.*

Intuitively, this is because the dependency constraint is the only constraint to observe during rule updates in TCAM, and the DAG precisely represents the minimum set of dependencies. Due to the space limitation, the proof is provided in a separate technical report [22].

Claim 2. *With DAG, the back-end can generate a flow table without obscured rules and floating rules.*

Through a simple topological scanning, *RuleTris* can eliminate two types of redundant rules generated during modular composition, i.e., the rules obscured by higher priority rules (or *obscured rules*) and the rule having the same actions with lower priority but more general rules (or *floating rules*).

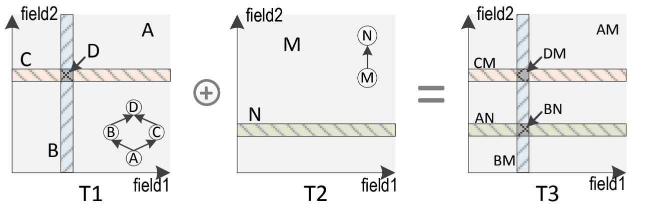
IV. Front-end Compiler

RuleTris front-end is an incremental composition compiler that compiles forwarding policy updates from SDN applications into rule updates and DAG updates for data-plane flow tables. State-of-the-art incremental compilation technique allows us to compile rule updates with integer priority in a few milliseconds [6]. However, the brute-force way to extract DAG from prioritized flow tables has the high time complexity [7], [13]. In practice, it can consume minutes in processing a flow table with a few thousand rules.

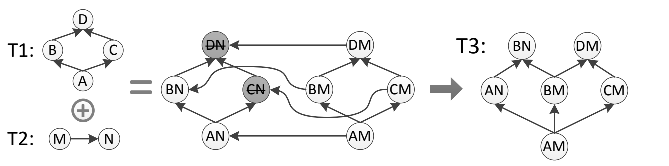
Alternatively, we choose to maintain the DAG along with the compilation process. The idea was first introduced in our previous short paper [7]. In this section, we extend the NetKAT-specific DAG preservation algorithm into an incremental and compiler-generic front-end by exploiting efficient data structures and algorithms. We first give the background on the modular composition (§IV-A). Then, we show how we build the DAG along with the composition with linear time complexity (§IV-B). Finally, we present the incremental techniques to further accelerate the compilation of DAG updates (§IV-C).

A. Modular Composition Basics

The ultimate goal of a composition compiler is to combine multiple member policies (or flow tables) into a single result policy. To do so, the existing compilers use the composition configuration (e.g., $(A > B) + C$) to guide the recursive composition compilation. Then, for each composition operator, the compiler combines the two member flow tables (T_1 and T_2) into the result flow table (T_3) according to the semantic of the operator. For parallel and sequential operator, the compiler explicitly iterates over rule pair $(r_{1,i}, r_{2,j}) \in T_1 \times T_2$ in a descending priority order, and calculates the result rule with an operator-specific function $para(r_1, r_2)/seq(r_1, r_2) : R \times R \rightarrow R$, where R is the universe set of rules. For parallel operator, the function $para(r_{1,i}, r_{2,j})$ produces a result rule with the match by taking the intersection of $r_{1,i}.match$ and $r_{2,j}.match$ and with the actions by taking the union of $r_{1,i}.actions$ and $r_{2,j}.actions$. For sequential operator, the function $seq(r_{1,i}, r_{2,j})$ produces a result rule with the match by first applying $r_{1,i}.actions$ onto $r_{1,i}.match$ and then intersecting with $r_{2,j}.match$, and with the actions by taking the union of $r_{1,i}.actions$ and $r_{2,j}.actions$. For priority operator, the compiler simply stacks the rules in T_1 on top of T_2 by configuring rules in T_1 with higher priorities than rules in T_2. The reader can refer to previous policy compilers for detailed description of the composition process [15], [23].



(a) Flow space of rules.



(b) Rule dependency inference.

Fig. 3: Example 1 of dependency construction in parallel composition: cross-product and empty rule removal.

B. Preserving DAG during Composition

To construct the DAG during the process of a composition operator, the *RuleTris* compiler needs algorithms to infer the precise dependency relations in the result flow table from the operand DAGs.

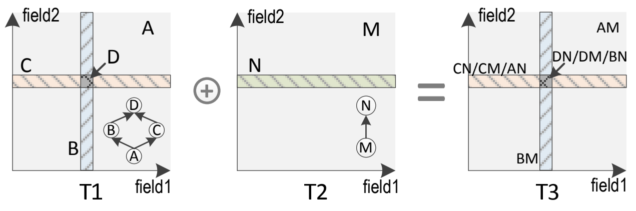
1) Parallel Composition: The parallel composition of T_1 and T_2 is calculated by taking cross-product of the operands. Similarly, the DAG of the result flow table is also calculated by taking the equivalent graph cross-product. Denoting two operand graphs as G_1 and G_2, the graph cross-product is defined intuitively as

1) The vertex set of $G_1 \times G_2$ is the set cross-product $V(G_1) \times V(G_2)$; and
2) There is a directed edge $\langle r_{1,i}, r_{2,m} \rangle \rightarrow \langle r_{1,j}, r_{2,n} \rangle$ in $G_1 \times G_2$ if and only if either ***i)*** $r_{1,i} = r_{1,j}$ and $r_{2,m} \rightarrow r_{2,n}$; or ***ii)*** $r_{2,m} = r_{2,n}$ and $r_{1,i} \rightarrow r_{1,j}$.

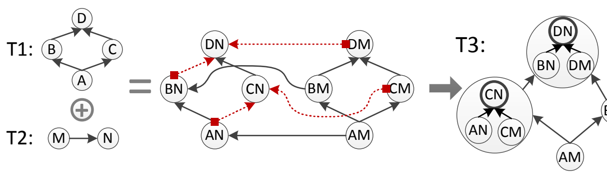
The correctness proof is intuitive. Consider rule r_1 depends on rule r_2, i.e., r_1 overlaps with r_2 and semantically r_2 has a higher priority than r_1. When we intersect both of them with a third rule r, the two result rules $(r_1 \cap r)$ and $(r_2 \cap r)$ still overlap, unless either of them has an empty match.

There are two cases that need special treatment. First, when the parallel composition of any rule pair results in an empty match, the corresponding vertex of this rule should not be added to the result DAG. For example, in Figure 3, we have two flow tables T_1 and T_2 taking the parallel composition. Specifically, T_1 contains four rules (A, B, C, D) and T_2 contains two rules (M, N). In the figure, the match space of the rules is visualized and the actions are omitted. To obtain the result DAG, the compiler first takes a cross-product of the operand DAGs. Then, the compiler crosses out the vertices of all the rules with empty match (DN and CN), and removes their adjacent edges from the DAG as well. Finally, the minimum DAG is obtained as shown on the right.

The second case is when two result vertices are adjacent but the corresponding rules have the same match. In this case, the higher priority rule entirely obscures the other one, so the latter becomes redundant. Although the redundant rules should



(a) Flow space of rules.



(b) Rule dependency inference.

Fig. 4: Example 2 of dependency construction in parallel composition: equivalent rule reduction.

be maintained within the compiler for the correctness of the future incremental rule removals, it is favorable to eliminate such redundancy in the current output.

We design a two-level nested graph structure to efficiently handle such redundancy. On the higher level, the compiler uses the rule match as the key to index the vertices, which we call *key vertices*. Therefore, multiple rules with the same match will fall into the same key vertex. If more than one rule is inserted into one key vertex, the dependency relations between those rules are recorded as a nested sub-graph. Within any key vertex, there must exist one single highest priority rule, because otherwise the composed flow table is ambiguous. When the compiler populates the flow table from the DAG, the highest priority rule is used to represent the key vertex, as it obscures all other rules in this key vertex.

Figure 4 shows an example of the parallel composition of T_1 and T_2. After the cross-product of the operand DAGs, we see several sets of vertices have the same match (e.g., BN, DN and DM). The compiler indexes these equivalent vertex sets with the nested graph data structure, which populates the flow table without redundant matches.

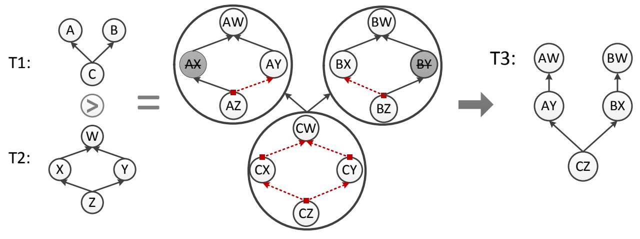
2) Sequential Composition: As shown in Section IV-A, the existing compilers calculate sequential composition of T_1 and T_2 in a two-level loop. The inner loop is similar to parallel composition. Each rule $r_{1,i}$ in T_1 produces a partial flow table $r_{1,i} > T_2$. For the outer loop, different partial flow tables are stacked by the priorities in T_1. This is because if $r_{1,i}.priority > r_{1,j}.priority$ in T_1, the partial flow table produced by $r_{1,i}$ will always be matched prior to that by $r_{1,j}$.

The DAG of the sequential composition can be also obtained through a similar two-level loop. For each rule $r_{1,i}$ in T_1, the DAG of the partial flow table $r_{1,i} > T_2$ is calculated by taking a cross-product, similar to the parallel composition. Then, the partial DAGs of the partial flow tables are stitched together according to the dependencies in T_1, i.e., if $r_{1,i} \rightarrow r_{1,j}$ in T_1, the partial DAG induced by $r_{1,i}$ is also dependent on the partial DAG by $r_{1,j}$.

Figure 5 shows an example of the sequential composition between T_1 and T_2. As shown in the middle of Figure 5(b), the partial DAGs in the three large circles are derived from the dependencies of T_2, e.g., $X \rightarrow W$ derives $AX \rightarrow AW$, $BX \rightarrow BW$ and $CX \rightarrow CW$. Meanwhile, the dependencies between partial DAGs are derived from the dependencies

	idx	Predicate	Actions
T1:	A	dst_port = 80	dst_ip = 1.0.0.0
	B	dst_port = 443	src_ip = 2.0.0.0
	C	*	Drop
T2:	W	src_ip=2.*.*.*, dst_ip = 1.*.*.*	Fwd(1)
	X	src_ip=2.*.*.*	Fwd(2)
	Y	dst_ip = 1.*.*.*	Fwd(3)
	Z	*	Drop
T3:	AW	src_ip=2.*.*.*, dst_port = 80	dst_ip = 1.0.0.0, Fwd(1)
	AY	dst_port = 80	dst_ip = 1.0.0.0, Fwd(3)
	BW	dst_ip = 1.*.*.*, dst_port = 443	src_ip = 2.0.0.0, Fwd(1)
	BX	dst_port = 443	src_ip = 2.0.0.0, Fwd(2)
	CZ	*	Drop

(a) Rule tables.



(b) Rule dependency inference.

Fig. 5: Example of sequential composition.

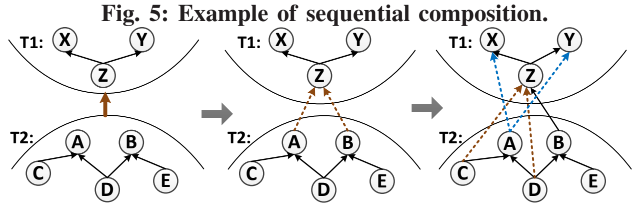


Fig. 6: Resolving mega dependency relations.

of T_1, e.g., $C \rightarrow A$ derives $(CW, CX, CY, CZ) \rightarrow (AW, AX, AY, AZ)$. Finally, after eliminating empty and redundant rules, we get the optimal flow table and its DAG of T_3 shown on the right of Figure 5(b).

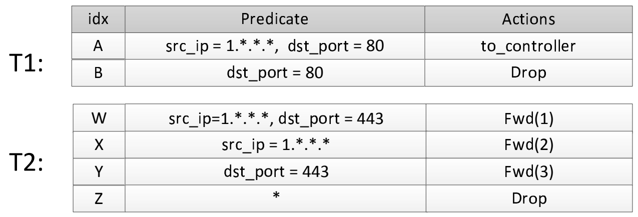
In some cases, the dependency relations between partial DAGs (or "mega" dependencies) need further refinement to produce a minimum set of the dependency relations. More precisely, we can create a mega edge from rule set A to rule set B, if for every rule pair $< a, b >$ $(a \in A, b \in B)$ we have either $a \rightarrow b$ or a is independent with b . We defer the detailed discussion to Section IV-B3.

3) Priority Composition: The priority composition of T_1 and T_2 is derived by stacking the flow tables by priority. Therefore, the priority composition of DAGs can be calculated by stitching the operand DAGs with a mega dependency relation from T_2 to T_1.

The challenge comes from resolving the mega dependency between T_1 and T_2 into dependencies between individual rules. Theoretically, the dependency relation between T_1 and T_2 does not necessarily derives the dependency between an arbitrary rule in T_1 and an arbitrary rule in T_2, since they may not overlap with each other. In order to obtain a minimum set of the dependency relations, the compiler needs to efficiently verify any possible rule dependency.

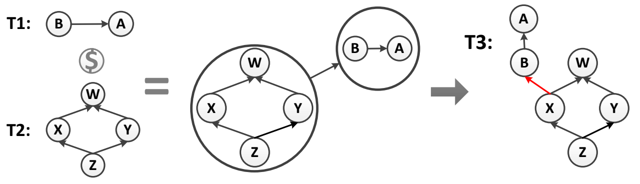
RuleTris compiler resolves the mega dependency relations with the following recursive procedure.

- First, the mega dependency from T_2 to T_1 is resolved to a set of tentative dependency relations from every sink vertex of T_2 to every source vertex of T_1. For example, in Figure 6,



	idx	Predicate	Actions
T1:	A	src_ip = 1.*.*.*, dst_port = 80	to_controller
	B	dst_port = 80	Drop
T2:	W	src_ip=1.*.*.*, dst_port = 443	Fwd(1)
	X	src_ip = 1.*.*.*	Fwd(2)
	Y	dst_port = 443	Fwd(3)
	Z	*	Drop

(a) Rule tables.



(b) Rule dependency inference.

Fig. 7: Example of priority composition.

the mega dependency relation is resolved to tentative edges $A \rightarrow Z$ and $B \rightarrow Z$.

- Then, for each tentative dependency edge $r_2 \rightarrow r_1$, the compiler explicitly checks whether the matches of the two rules r_1 and r_2 overlap. If so, edge $r_2 \rightarrow r_1$ is put into the result DAG. Otherwise, the compiler recursively generates tentative edges as follows.
  - For every predecessor of r_2, say r_3, if edge $r_3 \rightarrow r_1$ does not exist in the DAG already, the compiler adds it to the set of tentative edges, as r_3 has a more general match than r_2 and may overlap with r_1. In Figure 6, assuming A and Z do not overlap, the compiler adds $C \rightarrow Z$ and $D \rightarrow Z$ as tentative edges (red dashed edges).
  - For every successor of r_1, say r_4, if edge $r_2 \rightarrow r_4$ does not exist already, and meanwhile $r_1.match$ is not *strictly more general* than $r_4.match$ (meaning $r_1.match - r_4.match \neq \emptyset$ in flow space), the compiler also adds the edge $r_2 \rightarrow r_4$ to the set of tentative edges. This is because r_2 may overlap with r_4 on the excessive flow space $r_1.match - r_4.match$. In Figure 6, the compiler adds $A \rightarrow X$ and $A \rightarrow Y$ as tentative edges (blue dashed edges).
- The compiler continues resolving until the set of tentative edges is empty.

The complexity of the algorithm is bounded by the diameter of the partial DAG, which is typically small.

Finally, Figure 7 shows an example of the priority composition between T_1 and T_2. The compiler first adds a mega edge between the DAGs of T_1 and T_2. Then, the mega edge is resolved to a tentative edge from W to B. Because W does not overlap B, this tentative edge sprouts to tentative edges $X \rightarrow B$ and $Y \rightarrow B$. Note, $W \rightarrow A$ is not added as a tentative edge because $A.match$ is strictly smaller than $B.match$. Finally, edge $X \rightarrow B$ is added to the result DAG.

C. Incremental Compilation

Ideally, when processing a rule update, the composition compiler should only recompile the rules and the partial DAG that change during the update. We observe that most part of a DAG will not change during a rule update, which indicates the opportunity of dramatic performance improvement over recompilating from scratch.

RuleTris's incremental compilation technique is built on top of existing incremental composition technique. Previous study [6] proposes an efficient indexing structure for flow tables, which allows the compiler to efficiently find the rules that overlap with a target rule. *RuleTris* employs this technique to avoid redundant computation.

The key technique *RuleTris* introduces is the mechanism to compile DAG update. Upon the arrival of a rule update, the *RuleTris* compiler calculates the delta DAG as follows.

Rule insert. Consider a composition of T_1 and T_2. When the compiler receives a rule insert r_1 with the dependency change in T_1, the compiler first computes all the additional rules to be added in result similar to CoVisor. For parallel and sequential composition, it does so by looking up T_2's index for the rules that overlap with r_1, and apply composition function $para(r_1, r_2)/seq(r_1, r_2)$. For priority composition, r_1 is simply inserted into the result flow table.

Then, the compiler calculates the changes in the DAG of T_3. It adds vertices representing each inserted rules into the DAG. Further, the compiler handles dependency changes for the composition operators as follows:

- For parallel composition, the compiler takes a cross-product of the additional partial DAG in T_1 and the full DAG of T_2, and the result partial DAG is added to $T_3.graph$.
- For sequential composition, if r_1 belongs to the left operand (i.e., $T_1 > T_2$), the compiler composes r_1 with T_2 and adds the result partial graph to $T_3.graph$. The compiler also adds the edges associated with r_1 to $T_3.graph$ as mega dependency relations, and resolves them with the same procedure in Section IV-B2. If r_1 belongs to the right operand (i.e., $T_2 > T_1$), the compiler composes every rule in T_2 with r_1, and adds the result partial graph to $T_3.graph$. The compiler also resolves the mega dependencies in $T_3.graph$, since r_1 may change the actual edges those mega edges are resolved to.
- For priority composition, the compiler first adds the edges associated with r_1 to $T_3.graph$, and then resolves the mega dependency relation created by the priority operator.

RuleTris further accelerates the above graph compositions with the rule indexing structure. When taking a partial cross-product or sequential composition, the compiler only processes the partial DAG of T_2 whose rules overlap with r_1, because composing r_1 with any rules not overlapping it will result in an empty rule.

Rule delete. When a rule is deleted in a member flow table, all the rules that are composed from the deleted rule are to delete in the result flow table. If a deleted rule has both predecessors and successors in the DAG, the compiler will add tentative edges from every rules in the predecessor set to every rules in the successor set. Then, the compiler verifies the tentative edges in the same way as in Section IV-B3.

Rule modification. *RuleTris* handles rule modification equivalently as one delete plus one insert.

V. Back-end Optimizer

The DAG and DAG updates generated by *RuleTris* front-end are eventually exploited by *RuleTris* back-end to conduct optimization to TCAM updates. *RuleTris* currently has three back-end optimizers: update scheduler, duplication eliminator and CacheFlow manager. With them, *RuleTris* can provide guarantees to conduct rule updates with minimum number of TCAM moves, to compile minimum-size flow tables with no redundant rules and to provide support for efficient rule caching hierarchy.

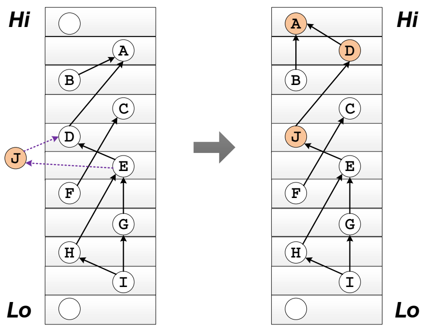


Fig. 8: Example TCAM move optimization.

A. Update Scheduler

The update scheduler exploits the DAG to optimize the rule update process in the TCAM. As described in Section II, TCAM can be viewed as a large array of rules that conducts efficient parallel lookup. When there are entries conflicting with each other on the match patterns, the entry located on the highest physical address wins. As a result, the switch firmware must maintain a correct ordering of rules during TCAM update.

Typically, the switch firmware works as follows. Upon the arrival of a rule insert, the firmware first checks the dependency relations (usually in the form of priority) with the layout of existing rules and looks for the range of locations that satisfy the dependency requirements. Then, it checks if there are empty slots within that range. If so, it picks a slot and writes the new rule in it. Otherwise, the firmware has to move the existing rules for an extra slot.

Integer priority value provides a poor clue of actual rule dependencies, and leads to massive redundant TCAM moves. *RuleTris* update scheduler exploits the DAG to optimize the TCAM updates. The *RuleTris* update scheduler first checks if there is an empty slot that satisfies the dependency constraints of the new rule. If so, the new rule is written to the slot. Otherwise, the update scheduler calls Algorithm 1 to search for an entry moving chain, which starts with the new rule and ends with an empty slot (e.g. $J \rightarrow D \rightarrow A \rightarrow Slot_{top}$ in Figure 8). Finally, the new rule is inserted by moving every rule in the moving chain one slot downstream. The optimality proof of Algorithm 1 can be found in a separate technical report [22].

For example in Figure 8, Rule J is to be inserted and its relative dependency is shown with the dotted arrows. The scheduler first finds the inserted location range between D and E, which has no slot available. Next, the scheduler looks for the nearest slots, which are located on the top and bottom of the figure. Then, the scheduler searches for moving chains, which are $J \rightarrow D \rightarrow A \rightarrow Slot_{top}$ on the upper side and $J \rightarrow E \rightarrow F \rightarrow Slot_{bottom}$ on the lower side. Since the number of entry moves are the same, a final update decision is picked on a random basis.

B. Redundancy Eliminator

The redundancy eliminator uses the DAG to remove two types of redundant rules:

1) **Obscured rules.** If a rule is entirely obscured by higher priority rules, no data plane packet will match this rule.
2) **Floating rules.** Consider two rules immediately adjacent in DAG. If they share the same actions and the lower-priority rule has a more general match than the higher-priority one, the higher-priority rule is redundant because removing it does not change the data plane behavior of the flow table.

ALGORITHM 1: SHORTEST MOVING CHAIN SEARCH.

```
Input : Rule DAG $< V, E >$, TCAM layout $f_r : Addr. \rightarrow V$, New
        rule to insert $r_{insert}$
Output: Shortest entry moving chain
$r_{succ} \leftarrow \arg\min_{<r_{insert}, r> \in E} r.addr$ /* $r_{insert}$'s lowest successor */
$r_{pre} \leftarrow \arg\max_{<r, r_{insert}> \in E} r.addr$ /*$r_{insert}$'s highest predecessor*/
$d_{succ}(d_{pre}) \leftarrow$ the closest empty slots from $r_{succ}$ ($r_{pre}$)
for $i \leftarrow d_{pre}$ to $d_{succ}$ do
    $f_r(i).move \leftarrow MAX\_INT$ /* initiation */
for $i \leftarrow r_{pre}.addr$ to $r_{succ}.addr$ do
    $f_r(i).move \leftarrow 1$ /* base cases */
    $f_r(i).prev \leftarrow r_{insert}$
for $i \leftarrow r_{pre}.addr + 1$ to $d_{succ}.addr - 1$ do
    /* Calculate the highest location $r_{curr}$ can be moved to */
    $r_{curr} \leftarrow f_r(i)$, $loc_{hi} \leftarrow d_{succ}.addr$
    foreach $r_{next}$ in $\{r| < r_{curr}, r > \in E\}$ do
        $loc_{hi} \leftarrow min(r_{next}.addr, loc_{hi})$
    /* Update backtrack states */
    for $j \leftarrow r_{curr}.addr + 1$ to $loc_{hi}$ do
        if $f_r(j).move > r_{curr}.move + 1$ then
            $f_r(j).move \leftarrow r_{curr}.move + 1$
            $f_r(j).prev \leftarrow r_{curr}$
for $i \leftarrow r_{succ}.addr - 1$ downto $d_{pre}.addr + 1$ do
    $r_{curr} \leftarrow f_r(i)$, $loc_{lo} \leftarrow d_{pre}.addr$
    foreach $r_{next}$ in $\{r| < r, r_{curr} > \in E\}$ do
        $loc_{lo} \leftarrow max(r_{next}.addr, loc_{lo})$
    for $j \leftarrow r_{curr}.addr - 1$ downto $loc_{lo}$ do
        if $f_r(j).move > r_{curr}.move + 1$ then
            $f_r(j).move \leftarrow r_{curr}.move + 1$
            $f_r(j).prev \leftarrow r_{curr}$
if $d_{succ}.move < d_{pre}.move$ then
    return the backtrack path from $r_{insert}$ to $d_{succ}$
else
    return the backtrack path from $r_{insert}$ to $d_{pre}$
```

RuleTris redundancy eliminator conducts one-time scan in a topologically decreasing order to remove the above types of redundant rules. Specifically, for each rule visited, the redundancy eliminator accumulates the match with a flow space union. If a visited rule is entirely obscured by the current accumulated match, it is a obscured rule and should be removed. If a visited rule has the same actions with any of its predecessors, and its match is narrower than the predecessor, it is a floating rule and should be removed.

C. CacheFlow Manager

CacheFlow manager maintains a hierarchy of rule caches and helps scale up the size of physical flow tables with larger but slower flow table implementations, such as in SRAM. This technique was proposed in previous work [13]. The key idea is to maintain the correct dependency of the partial flow table in high-speed cache by inserting “cover-set” rules that redirect data plane packets to low-speed matching hardware. We refer the reader to the original paper for details.

VI. Implementation

We implement *RuleTris* front-end composition compiler with 5K lines of Java code. For comparison, we also implement a baseline composition compiler, which recompiles from scratch for each update, and the CoVisor composition compiler [6], which does efficient incremental composition using the priority algebra.

We implement *RuleTris* back-end optimizers by extending the firmware on the ONetSwitch with 3K lines of C code [16]. ONetSwitch is hardware based all programmable SDN switch which allows us to fully amend the firmware for RuleTris. We extend OpenFlow v1.3 protocol with DAG support using experimenter messages. The extension can carry both full DAGs and incremental DAG updates from the front-end to the firmware back-end. In the experiments, *RuleTris* composition compiler uses the extended OpenFlow to talk to *RuleTris* back-end firmware, while the baseline compiler and the CoVisor compiler uses the original ONetSwitch firmware with full OpenFlow v1.3 support.

VII. Evaluation

We evaluate the effectiveness and runtime overhead of *RuleTris* with both hardware and emulation experiments.

A. Methodology

a) Experiment Setup: We evaluate *RuleTris* under three scenarios. The first two evaluate the rule update efficiency of *RuleTris* with parallel and sequential compositions. The third one evaluates the rule swapping efficiency with the CacheFlow back-end. In each scenario, we conduct hardware experiments using aforementioned ONetSwitch with a 256-entry TCAM flow table, and stress *RuleTris* with larger flow table updates through firmware emulation. Except otherwise noted, we maintain a reasonably high TCAM load factor of 0.90 in the emulation experiments.

We run all composition compilers on top of Ryu controller [24]. The front-end compilation and the back-end emulation are done on a Linux workstation with 4 cores at 2.8GHz and 8GB memory.

In the experiments, we compare *RuleTris* with the following composition compilers.

Baseline. The baseline compiler recompiles the new flow table from scratch for every rule update and assigns sequential priority values to the new flow table.

CoVisor. The CoVisor compiler conducts incremental compilation to rule updates with the efficient rule indexing structure. It assigns priority to new rules using a convenient algebra that prevents reprioritizing.

b) Dataset:

- **L3-L4 monitoring + L3 router.** In this scenario, the L3-L4 monitoring app collects flow statistics in parallel with a L3 router conducting IP-based forwarding. We generate monitoring rules using network filter generation tool ClassBench [25] with the firewall configuration. L3 router rules are also generated using ClassBench, but with the IP chain configuration.
- **L3-L4 NAT $>$ L3 router.** L3 router rules are generated similar as above. L3-L4 network address translation (NAT) tables are randomly generated based on the IP addresses and TCP/UDP ports of the router rules.
- **CacheFlow rule swapping.** CacheFlow picks a subset of rules from a full rule set to put in cache. In our experiment, the full rule set is a forwarding rule database with 1000 rules generated similar as previous L3 router rules. A set of rules is randomly selected to be installed in the TCAM as well as the necessary cover-set rules that ensure correct matching semantics. Then, a sequence of swap-in/swap-out operations is randomly generated to mimic the cache swapping behavior.

c) Metrics: In the following figures, the bars show the median, and the error bars show the 10th and 90th percentiles.

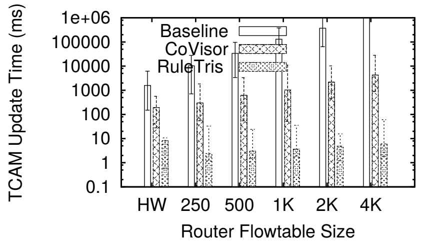
- **Compilation time.** The computation time to compile the rule update in the front-end.
- **Firmware time.** The computation time to calculate the update schedule from a priority-based or dependency-graph-based rule update in the switch firmware. In hardware experiments, this time is measured on the 800MHz ARM Cortex-A9 CPU on ONetSwitch by switch firmware. In the emulation experiments, the firmware time is measured on the workstation emulating the physical switch.
- **TCAM update time.** The actual time to conduct rule updates on the TCAM. Since TCAM moves are conducted sequentially and each TCAM move costs a fairly constant amount of time, we use the total number of moves times the average latency of a TCAM move (0.6ms) to estimate the TCAM update time in emulation experiments.

B. Experimental Results

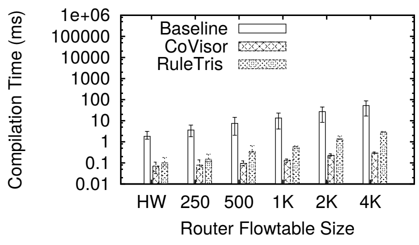
Figure 9 shows the results of L3-L4 monitoring + L3 router. In this experiment, we initiate L3-L4 monitoring table with 100 rules and L3 router with 250 to 4K rules to show how the overhead increases. We sequentially feed 1000 updates to compilers, each update contains one rule delete and one rule insert to the L3-L4 monitoring table. The size of L3 routers is set to 78 in the hardware experiment (first group) in order to fit the 256-entry TCAM.

The compilation time, firmware time and TCAM update time are shown in Figure 9(a), 9(b) and 9(c) respectively. The baseline compiler is by far the slowest regarding all three metrics. This is because it recompiles the flow table in every round with new priority value assignments, and thus generates a large amount of redundant rule updates that only modifies the rule priority. In the hardware experiment, *RuleTris* exhibits 20x faster total update time than CoVisor adding all three latency components together. And emulations indicate even greater differences. Among three latency components, TCAM update time contributes the most. *RuleTris* has the smallest TCAM update latency, which is fairly independent of flow table size. This is because *RuleTris* maintains the DAG that helps the firmware to calculate the optimal update schedule. Since CoVisor does not keep DAG, it is the fastest in compilation and firmware time, but spends 1 to 3 orders of magnitude more time on TCAM update. Note, the hardware experiment shows a higher firmware time than emulations because of the different capacity of the processors.

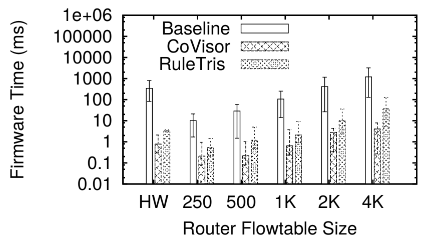
Figure 10 shows the result of L3-L4 NAT $>$ L3 router. Same as the previous experiment, we initiate L3-L4 NAT table with 100 rules and L3 router with 250 to 4K rules to show how the overhead increases. We sequentially feed 1000 updates to compilers, each update contains one rule remove from and one rule insert to the NAT table. The size of L3 routers is set to 126 in the hardware experiment. Again, we observe *RuleTris* exhibits about 20x faster total update time than CoVisor due to the time saved in the TCAM updates.



(a) TCAM Update Time.

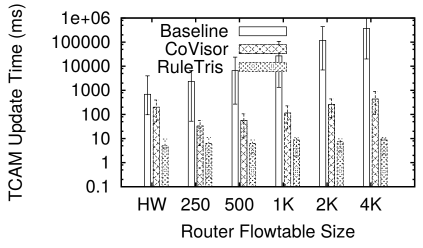


(b) Compilation Time.

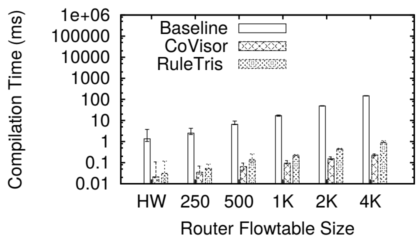


(c) Firmware Time.

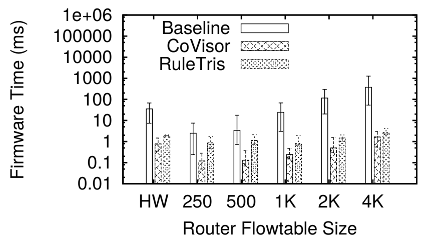
Fig. 9: Rule update overhead of L3-L4 monitoring + L3 router. The first group (HW) is hardware experiment results; and the rest are emulation results.



(a) TCAM Update Time.



(b) Compilation Time.

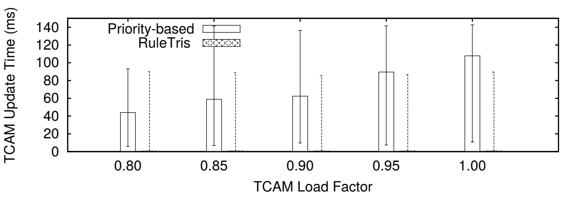


(c) Firmware Time.

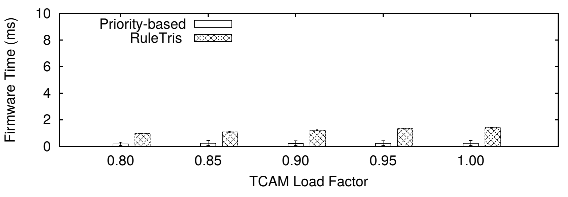
Fig. 10: Rule update overhead of L3-L4 NAT $>$ L3 router.

Figure 11 shows the result of CacheFlow rule swapping. In this experiment, we create a two-level CacheFlow with the physical switch as the first level. We vary the load factor of the first level from 0.8 to 1.0. We compare the rule swapping efficiency of *RuleTris* with the priority-based update firmware. We initiate the CacheFlow manager with a thousand L3 forwarding rules. We randomly select 205 to 256 rules (according to the load factor) to install into the first level. We sequentially feed 1000 updates to the CacheFlow manager, each update contains one rule delete and one rule insert to the TCAM table.

The TCAM update time and firmware time are shown in Figure 11(a) and 11(b) respectively. As expected, *RuleTris*'s DAG based updates shows a dominant advantage over the priority-based updates. The median of *RuleTris* TCAM update time ranges from 0.6 to 1.2 milliseconds, whose bars can be barely seen in the figure. In contrast, priority-based updates costs 40 to 100 milliseconds per rule swapping, and the per-operation cost increases significantly with the TCAM load factor. The long tail of the *RuleTris* update time is due to some of the swap-in rules that have dense dependency with the rules in cache, which leads to multiple entry moves in TCAM.



(a) TCAM Update Time.



(b) Firmware Time.

Fig. 11: Rule update overhead of single rule swap with CacheFlow. Results are from hardware experiments.

VIII. Discussion

Multiple tables. *RuleTris* currently optimizes updates to a single flow table. Switches typically have multiple tables. Depending on the order of execution of the tables, we can further minimize the rule updates. For example, if we have two TCAM tables in a pipeline, the dependencies between the two modules in a sequential composition can be decoupled by placing the first one in the first TCAM and the second module in the second TCAM. However, the number of tables in a hardware switch is limited. *RuleTris* can support more module compositions than the number of physical flow tables. We leave the effective distribution of rules to multiple flow tables to our future work.

Hardware specific optimizations. Tango [26] and Mazu [18] have shown that different switches can have very different latency behavior depending on the order of rule updates. For example, given two ordering of a batch of rules, one in increasing priority, the other in decreasing priority. One switch has a much lower latency for the first order. Techniques [26], [18] proposed to exploit hardware behavior can be usefully combined with *RuleTris*.

Network update cooperation. *RuleTris* considers per switch flow table updates independently. Coordination among several switches can be combined with *RuleTris* to further reduce the number of updates [5], [27], [28], [29], [30].

IX. Related Work

Modular composition. Several recent SDN policy languages and controllers (e.g., Frenetic [21], NetCore [23], NetKAT [15], Pyretic [14]) support modular composition. Generally, they take high-level policies and generate flow tables that fulfill the semantics of the sequential and parallel composition. A recent work proposes CoVisor [6], a controller hypervisor that assigns priority value with a convenient algebra without changing the priority of existing rules. Although CoVisor significantly reduces the number of rule updates, it does not optimize the cost of individual rule updates.

Further, CoVisor assumes that the guest controllers are able to produce optimal updates, which is still a challenging problem for the guest controllers. In contrast, *RuleTris* minimizes both the number of rule updates and the cost of individual updates in TCAM, and it also works with incremental-agnostic applications/controllers.

Modular composition optimization. Our previous short paper [7] first proposed to preserve rule dependency during compilation. It sketched a solution framework with a compiler-specific dependency preserving algorithm and a heuristic-based priority assignment strategy. *RuleTris* extends the idea with two fundamental improvements. First, *RuleTris* proposes a compiler-generic dependency preserving algorithm with incremental compilation capacity in the front-end. Second, the back-end now uses rule dependency to minimize TCAM operations instead of rule priorities, leading to a significant reduction in actual TCAM update time.

Incremental TCAM update. Another related and well-explored topic is incremental TCAM updates. TCAM uses the physical location to encode the priority of entries [19]. During TCAM incremental update, TCAM controller must maintain a correct order of entries based on the limited knowledge of the entry dependency, causing moves of existing entries. Although many algorithms have been proposed to infer entry dependency and reduce the update cost [8], [9], [11], it remains computationally challenging to obtain the minimum dependency graph for a flow table with wildcard matching and multiple matching fields. In contrast, we achieve the update cost minimization through leveraging the minimum dependency information generated in policy composition.

Incremental compilation. Most SDN compilers do not support incremental policy compilation. In practice, they simply compile the new policies and replace the entire flow table of each switch. Maple [12] introduces tree-style abstraction to support incremental flow table compilation. However, Maple compiler does not support policy composition and it makes redundant priority updates due to the consecutively assigned priority values. *RuleTris* can be integrated into Maple to provide optimal TCAM updates.

CoVisor [6] assigns priorities that leads to an inefficient usage of priority value space with priority multiply, which in turn limits the number of controllers it can support. Also, the large number of priority levels assigned by CoVisor aggravates to slow rule updates of TCAM. In contrast, *RuleTris* discards priority values and use the DAG to represent rule dependency.

X. Conclusion and Future Work

Fast incremental flow table updates due to policy changes is critical to the agility of the SDN control plane. We identify the TCAM update latency in physical OpenFlow switches as one of the major bottleneck to fast flow table updates. We present the first end-to-end optimization framework, *RuleTris* that incrementally keeps DAG during policy compilation and exploits DAG for optimal TCAM updates. We fully implement *RuleTris* and demonstrate its optimality with both hardware experiments and emulations.

Acknoledgement

This material is based upon work supported in part by the NSF under grant No. CNS-1219116 and by NSFC under grant No. 61472359, 61472209 and 61402404. The statements made herein are solely the responsibility of the authors.

References

[1] B Niven-Jenkins, D Brungard, M Betts, N Sprecher, and S Ueno. Requirements of an MPLS Transport Profile. *RFC 5654 (Proposed Standard), IETF*, 2009.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[3] Solution Brief: SDN Security Considerations in the Data Center. http://bit.ly/1KQGUXg.

[4] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What You Need to Know About SDN Flow Tables. In *Passive and Active Measurement*, pages 347–359. Springer, 2015.

[5] Xin Jin, Hongqiang Liu, et al. Dynamic scheduling of network updates. In *ACM SIGCOMM*, 2014.

[6] Xin Jin, Jennifer Gossels, et al. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *USENIX NSDI*, 2015.

[7] Xitao Wen, Chunxiao Diao, et al. Compiling Minimum Incremental Update for Modular SDN Languages. In *HotSDN*, 2014.

[8] Jan Van Lunteren and Ton Engbersen. Fast and scalable packet classification. *IEEE J-SAC*, 21(4):560–571, 2003.

[9] Tania Mishra et al. DUOS–Simple Dual TCAM Architecture for Routing Tables with Incremental Update. In *IEEE ISCC*, 2010.

[10] Haoyu Song and Jonathan Turner. Fast Filter Updates for Packet Classification using TCAM. In *IEEE GLOBECOM*, 2006.

[11] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for tcams. *IEEE Micro*, 21(1):36–47, 2001.

[12] Andreas Voellmy, Junchang Wang, et al. Maple: simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

[13] Naga Katta et al. Infinite CacheFlow in Software-defined Networks. In *HotSDN*, 2014.

[14] Joshua Reich, Christopher Monsanto, et al. Composing Software Defined Networks. In *USENIX NSDI'13*.

[15] Carolyn Jane Anderson, Nate Foster, et al. NetKAT: Semantic Foundations for Networks. In *POPL*, 2014.

[16] Chengchen Hu, Ji Yang, et al. Design of All Programable Innovation Platform for Software Defined Networking. 2014.

[17] ONetSwitch45. http://bit.ly/1FmPfsH.

[18] Keqiang He et al. Mazu: Taming Latency in Software Defined Networks. Technical report, University of Wisconsin-Madison, 2014.

[19] Kostas Pagiamtzis and Ali Sheikholeslami. Content-Addressable Memory Circuits and Architectures: A Tutorial and Survey. *IEEE JSSC*, 41(3):712–727, 2006.

[20] Broadcom NL91024 Knowledge-Based Processor. http://bit.ly/1OkNHEC.

[21] Nate Foster, Rob Harrison, et al. Frenetic: A network programming language. In *ACM SIGPLAN*, volume 46, 2011.

[22] APPENDIX: RuleTris Back-end Update Scheduler Optimality Proof. http://bit.ly/1IFnxjj.

[23] Christopher Monsanto, Nate Foster, et al. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, volume 47, pages 217–230.

[24] Ryu OpenFlow Controller. http://osrg.github.io/ryu/.

[25] David E. Taylor et al. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007.

[26] Angelos Lazaris, Daniel Tahara, et al. Tango: Simplifying SDN Programming with Automatic Switch Behavior Inference, Abstraction, and Optimization. In *Proceedings of CoNext'14*, 2014.

[27] Chi-Yao Hong, Srikanth Kandula, et al. Achieving High Utilization with Software-driven WAN. In *ACM SIGCOMM*, 2013.

[28] Hongqiang Harry Liu et al. zUpdate: Updating Data Center Networks with Zero Loss. *SIGCOMM CCR*, 43(4):411–422, August 2013.

[29] Mark Reitblatt, Nate Foster, et al. Abstractions for Network Update. In *ACM SIGCOMM*, 2012.

[30] Naga Praveen Katta, Jennifer Rexford, et al. Incremental consistent updates. In *HotSDN*, 2013.