DOI: 10.19729/j.cnki.1673-5188.2018.04.004
http://kns.cnki.net/kcms/detail/34.1294.TN.20181121.1527.004.html, published online November 21, 2018

Optimization Framework for Minimizing Rule Update Latency in SDN Switches

CHEN Yan [1,2], WEN Xitao[3], LENG Xue [1], YANG Bo[4], Li Erran Li [5], ZHENG Peng [6], and HU Chengchen[6]
(1. College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China;
2. Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA;
3. Google Inc., Mountain View, CA 94043, USA;
4. Microsoft, Shanghai 200000, China;
5. Uber Technologies Inc., San Francisco, CA 94103, USA;
6. School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China)

Abstract

Benefited from the design of separating control plane and data plane, software defined networking (SDN) is widely concerned and applied. Its quick response capability to network events with changes in network policies enables more dynamic management of data center networks. Although the SDN controller architecture is increasingly optimized for swift policy updates, the data plane, especially the prevailing ternary content-addressable memory (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 calculates the TCAM update schedules that minimize TCAM entry moves, which are the main cause of TCAM update inefficiency. In evaluation, RuleTris achieves a median of <12 ms and 90-percentile of < 15ms the end-to-end perrule update latency on our hardware prototype, outperforming the state-of-the-art composition compiler CoVisor by ~ 20 times.

Keywords

SDN; SDN-based cloud; network management; access control; unauthorized attack

1 Introduction

As a new network architecture proposed ten years ago, software defined networking (SDN) has been well researched in both academia and industry. The main reason that SDN is so concerned is its ability to dynamically change the network states in response to the global view. However, the response time to the network events determines how many new network applications can become practical. For example, the carrier network has a strict 50 ms requirement for failure recovery [1], entailing a 10 ms to 25 ms delay budget for implementing the rerouting rules. Traffic engineering in data centers has a delay budget as short as 100 ms for the entire control loop [2], leaving less than 20 ms delay budget for implementing flow rules. The 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 processing delay of a user request can be roughly divided into four parts: latency inside the controller, inside switches, and passing through the northbound and southbound communication channels. The transmission delay in the communication channel can be ignored and 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 33 ms to 400 ms 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

This work is supported by National Key R&D Program of China under Grant No. 2017YFB0801703 and the Key Research and Development Program of Zhejiang Province under Grant No. 2018C01088.

flow tables[1]. In addition, the measurement also finds that the switches can "periodically or randomly stop processing control plane commands for up to 400 ms", which further exacerbates the rule installation latency.

To reduce the latency inside switches, some existing works optimize the policy updates at different levels of the pipeline, however, 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 Static Random Access Memory (SRAM)/TCAM structure [8]–[10] or only applicable to single-field longest prefix matching [11].

Based on our research, the latency bottleneck within the TCAM-based SDN switches is introduced by policy update and the TCAM update latency is the single dominant factor of the rule update latency. Interestingly, although a single entry update 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.

In this paper, we present RuleTris, the first optimization framework for modular composition achieving minimum rule size and optimal rule update cost in TCAM. Our study reveals that the minimum dependency graph (DAG) [10], [12], [13] is the key information towards optimal rule updates. Compared with rule priorities, the DAG is a more fundamental and precise representation of the rule dependency. The DAG not only minimizes the number of rule updates sent to switches, but also minimizes the cost of individual rule updates by cutting 90% to 99% of TCAM micro operations.

As depicted in **Fig. 1**, RuleTris is consisted of a front-end and a back-end. The 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 Net Kleene Algebra with Tests (NetKAT) policy compiler [15] to reduce the 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 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 design an efficient and generic front-end policy compiler that generates DAG along with flow table compositions.

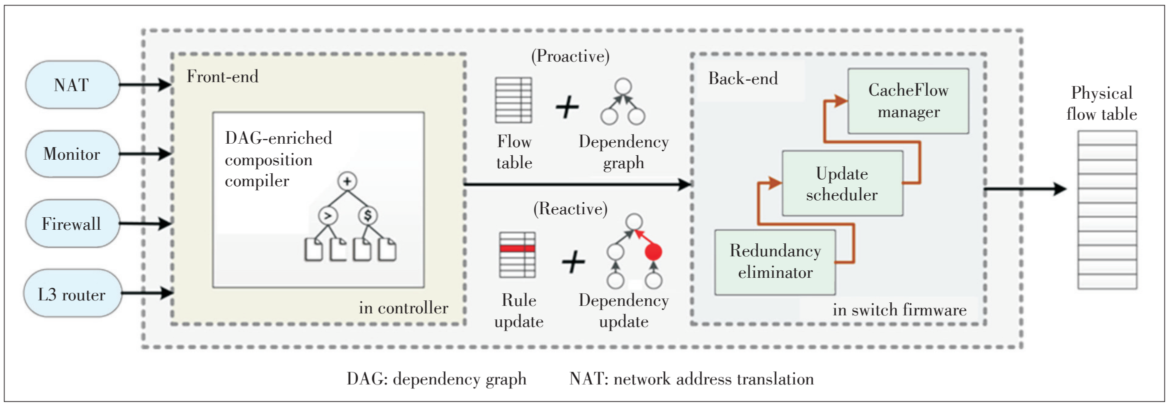


Figure 1. Overview of RuleTris optimization framework.

[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.

The front-end achieves efficiency by exploiting the dependency implications of composition operators with the specialized data structures. Our front-end further speeds up the compilation by incrementally compiling every rule update. What's more, the front-end compiler is generic to all SDN policy languages that employ policy compositions, and is guaranteed to produce the minimum DAG.

4) 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 can be 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 data-plane programmable hardware-based ONetSwitch [16], [17]. Through hardware evaluation, we demonstrate that RuleTris achieves a median of <12 ms and 90-percentile of <15 ms the per-rule update latency, outperforming the state-of-the-art composition compiler CoVisor deployed on the same hardware switch by ~ 20x. Our large scale emulation indicates even greater speedup on larger TCAM size.

We give background and related work in Section 2, followed by an overview in Section 3. We describe the front-end design in Section 4, priority value assignment algorithm in Section 5 and back-end design in Section 6. We present our implementation in Section 7, evaluation in Section 8, provide discussions on future topics in Section 9 and conclude in Section 10.

2 Background and Related Work

2.1 Background

1) 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. Upon the arrival of a new rule, the switch firmware may have to move many existing entries 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.

2) 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 DAG 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.

3) 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], [20]. In this paper, we compose applications with three composition operators: parallel operator, sequential operator, and priority operator. The parallel operator (+) creates the illusion that multiple applications to independently process the same traffic. The sequential operator (>) allows one application to process the traffic before another. The priority operator ($) gives one application the priority to act on a subset of the 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.

2.2 Related Work

1) Modular Composition

Several recent SDN policy languages and controllers (e.g., Frenetic [20], NetCore [21], 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.

2) 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.

3) Incremental TCAM Update

Another related and well-explored topic is incremental TCAM updates. TCAM uses the physical location to encode the priority of entries, with lower addresses (or higher addresses, depending on specific implementation) receiving higher priority [19]. During TCAM incremental update, TCAM controller must maintain a correct order of entries based on the limited knowledge of the entry dependency, which may cause 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.

4) Incremental Compilation

Most compilers, except Maple [12], do not support incremental policy compilation. In practice, they simply compile the new policies and replace the entire flow table of each switch. On the other hand, although Maple does not support policy composition, it introduces tree-style abstraction to support incremental flow table compilation. However, Maple compiler still 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 lead 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.

3 Overview of RuleTris

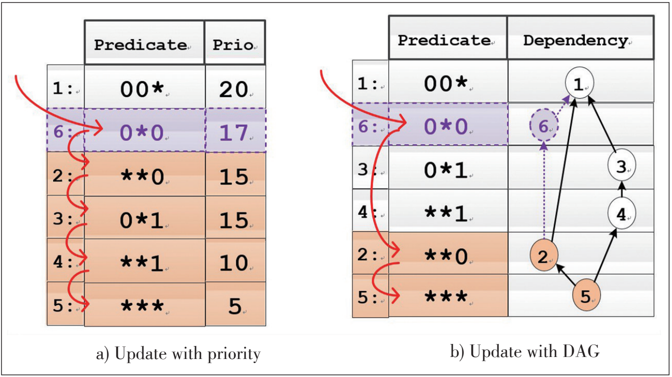
In this section, we first motivate the necessity of the DAG with an example in Section 3.1. We then depict RuleTris optimization framework in Section 3.2, followed by the optimality claims in Section 3.3.

3.1 Benefits of DAG

The key idea for RuleTris to generate minimum update is to represent rule dependency using DAG instead of rule priority until the controller finally compares old and new flow tables. Intuitively, a complete and minimum DAG as the intermediate representation provides the controller the maximum freedom to reuse the priority values of the existing rules, so that the generated updates contain no redundant priority changes.

Generally, optimally updating TCAM tables in physical switches requires a minimum DAG. In implementing a rule update in the TCAM table, integer priority values provide complete dependency information and thus can be used to generate semantically correct update schedule. For example, in **Fig. 2**, Rule 6 is to be added to the flow table. As shown in Fig. 2a, 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 very end, each of Rule 2 through Rule 5 has to be moved one slot down in order 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



▲ **Figure 2. An example rule insert in a TCAM table. The original TCAM table has five entries (Rules 1-5) and one empty slot in the end. Rule 6 needs to be inserted between Rules 1 and 2. In 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 b), however, the DAG indicates the newly inserted Rule 6 has no dependency with Rules 3 and 4, therefore only Rules 2 and 5 need to be moved.**

constraints. During the rule update, these redundant dependencies lead to unnecessary TCAM 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 Section 3.3). For example, Fig. 2b 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].

3.2 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 Fig. 1. RuleTris optimization framework is comprised of the front-end composition compiler and the back-end optimizers.

1) 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. The RuleTris composition compiler interfaces with applications or controllers, accepting their proactive or reactive modification of the network policies. Similar with other composition compilers, the RuleTris composition compiler is configured by the administrator to compose the application policies into a single policy implementation for physical network devices. Inspired by previous works, RuleTris allows policy composition with parallel operator (+), sequential operator (>) and priority operator ($) with similar semantics as previous modular composition compilers [6], [7], [14], [20].

Except 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.

2) 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 minimum-size update schedule to implement rule updates in TCAM tables. The redundancy eliminator removes all the semantically redundant rules. The CacheFlow manager manages multiple-level rule cache structure and conducts rule eviction guided by the DAG [13]. The RuleTris back-end directly generates sequence of TCAM entry moves.

3) Front-End/Back-End Communication

In this paper, we assume the RuleTris back-end is placed in the firmware of physical switches. The front-end to back-end communication is carried through the control channel, e.g., the OpenFlow protocol. RuleTris extends OpenFlow protocol with a DAG extension using the customizable experimenter message, so as to allow the protocol messages to carry DAGs or DAG updates together with flow modification/delete messages. Alternatively, RuleTris back-end can also be co-located with the front-end. In this way, no special front-to-back channel for DAG is necessary but the control channel needs to be extended to expose the TCAM internal layout.

3.3 Optimality Guarantees

RuleTris provides several optimality guarantees with the help of DAG and proper back-end optimizers. We show how the optimality is achieved in Section 6.

Claim 1: With DAG, the back-end can generate a flow table without obscured rules and floating rules.

Through a simple topological scanning, RuleTris can eliminate all the redundant rules generated during modular composition, including 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).

Claim 2: With DAG, the back-end can generate the minimum number of entry moves that correctly implements a specific rule update in a TCAM.

This is because the dependency constraint is the only constraint to observe during rule updates in TCAM, and the DAG precisely provides the minimum set of dependency constraints regarding a rule update. The proof is provided in the Appendix.

4 Front-End Compiler

The 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 some background on the modular composition (Section 4.1). Then, we show how we build the DAG along with the composition with linear time complexity (Section 4.2). We present the incremental techniques to further accelerate the compilation of DAG updates (Section 4.3).

4.1 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[2] [15], [21].

4.2 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. In addition, we also need efficient data structures to keep the DAGs and the DAG updates.

4.2.1 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 $V(G_1)\times V(G_2)$;

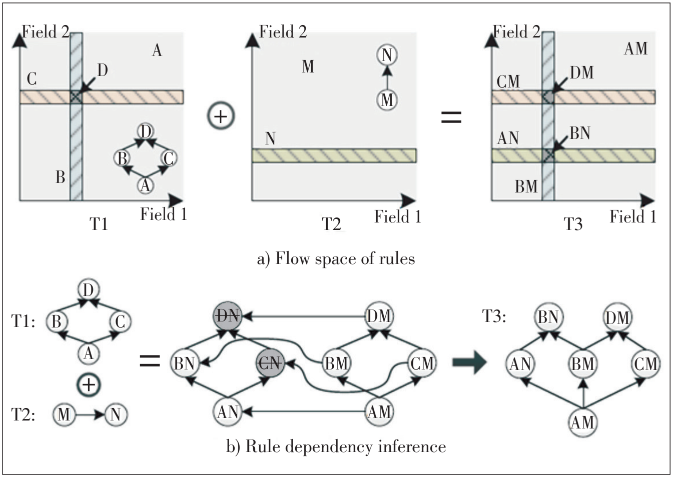
2) There is a directed edge $\langle r_{1,i},r_{2,m}\rangle\rightarrow\langle r_{1,j},r_{2,n}\rangle$ in $G_1\times G_2$ if and only if either i) $r_{1,i}=r_{1,j}$ and $r_{2,m}\rightarrow r_{2,n}$; or ii) $r_{2,m}=r_{2,n}$ and $r_{1,i}\rightarrow r_{1,j}$.

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 with each other, 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 **Fig. 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 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



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

[2] We assume all flow tables have a default match-all rule with a pseudo "pass" action, which passes the packet to the next flow table composed with the priority operator or drop the packet if there is not the one.

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.

Fig. 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.

4.2.2 Sequential Composition

In Section 4.1, 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 $r_{1,i}.priority > r_{1,j}.priority$, 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 $r_{1,j}$.

Fig. 5 shows an example of the sequential composition between T_1 and T_2. As shown in the middle of Figure 5b, 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 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 Fig. 5b.

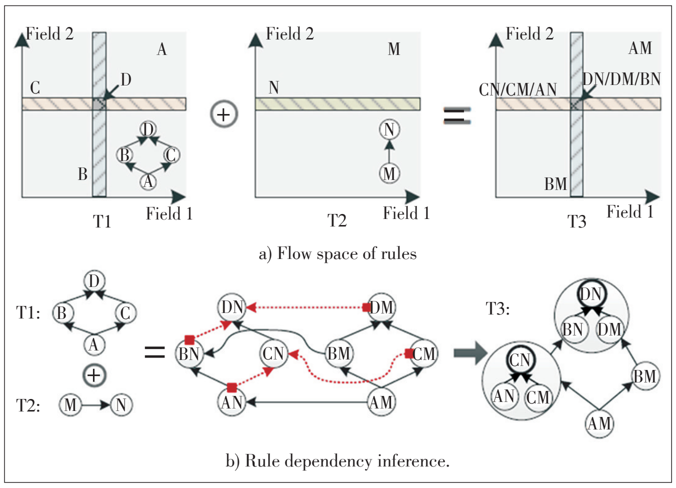
In some cases, the dependency relations between partial DAGs (or "mega" dependencies) need further refinement to produce a minimum set of the dependency relations. More precisely, we can create a mega edge from rule set A to rule set B, if for every rule pair $<a, b>$ ($a \in A$, $b \in B$) we have either $a \rightarrow b$ or a is independent with b. We defer the detailed discussion to Section 4.2.3.

4.2.3 Priority Composition

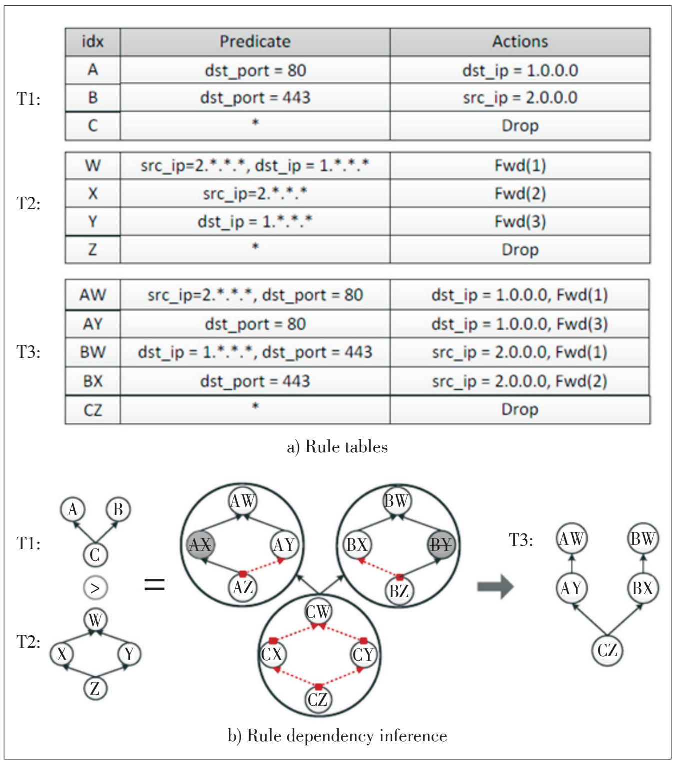
The priority composition of T_1 and T_2 is derived by stacking the flow tables by priority. Therefore, the priority composition of DAGs can be calculated by stitching the operand DAGs with a mega dependency relation from T_2 to T_1.

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 derive 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



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



	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

▲Figure 5. Example of sequential composition.

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, where source vertices (sink vertices) are defined as the vertices that has no incoming (outgoing) edges. For example, in **Fig. 6**, the mega dependency relation is resolved to tentative edges $A \rightarrow Z$ and $B \rightarrow Z$.

Then, for each tentative dependency relation (or 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. For example, in Fig. 6, assuming A and Z do not overlap, the compiler will add $C \rightarrow Z$ 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 $r_1.match$ — $r_4.match$, $\varnothing$ 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 $r_1.match - r_4.match$. For example, in Fig. 6, the compiler will also add $A \rightarrow X$ and $A \rightarrow Y$ as tentative edges (blue dashed edges).

In this way, the compiler continues resolving tentative edges until the set of tentative edges is empty.

Finally, **Fig. 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 C. Because W does not overlap C, this tentative edge sprouts to tentative edges $X \rightarrow C$ and $Y \rightarrow C$. Note, $W \rightarrow A$ is not added as a tentative edge because $A.match$ is strictly smaller than $B.match$. Finally, edge $X \rightarrow C$ is added to the result DAG.

4.3 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 recompilation from scratch.

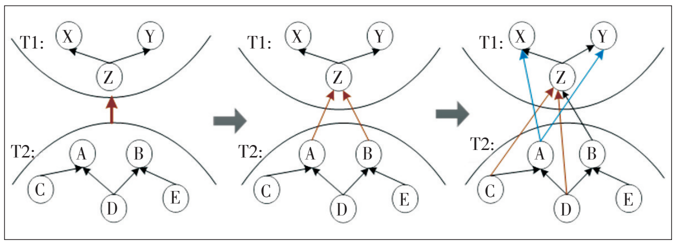
RuleTris' s incremental compilation technique is built on top of existing incremental composition technique. Previous study [6] proposes an efficient indexing structure for flow tables, which allows the compiler to efficiently find the rules that overlap with a target rule. RuleTris employs this technique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

The key technique RuleTris introduces is the mechanism to compile DAG update. Upon the arrival of a rule update with dependency change in the member policy, the RuleTris compiler calculates the delta DAG as follows.

1) 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 the rules inserted into the DAG. Further, the compiler handles dependency changes for the composition operators as follows:

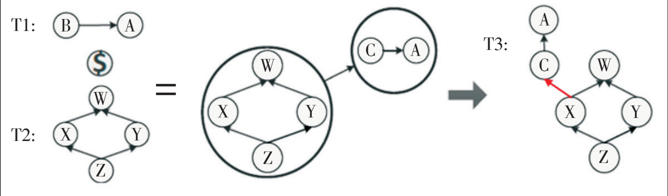
- For parallel composition, the compiler takes a cross-product of the additional partial DAG in T_1 and the full DAG of T_2, and the result partial DAG is added to $T_3.graph$.
- For sequential composition, if r_1 belongs to the left operand (i.e., $T_1 > T_2$), the compiler composes r_1 with T_2 and 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 4.2.2. 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 the mega edges are resolved to.



▲Figure 6. Resolving mega dependency relations.

	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

▲Figure 7. Example of priority composition.

- 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.

2) 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 4.2.3.

3) Rule modification

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

5 Assigning Priority Values

Another challenging step in RuleTris minimum update framework is to assign discrete priority values to the rules in the new flow table. The priority value assignment must observe both the dependency constraint and the integer priority constraint. The objective of this step is to reuse as many rule priorities as possible, so as to minimize the number of priority changes on existing rules. We formulate the optimization problem as follows.

5.1 Problem Formulation

The prioritizer takes as input a directed dependency graph with its vertices representing rules. There are two types of vertices. Some vertices are annotated as retained and each of them is associated with a priority value, which is an integer within a given range. The other vertices are annotated as new and they are not associated with any value. The output of prioritizer is a mapping from the vertex set to the set of priority values that preserves the dependency constraint, i.e., if there is an edge from Vertex A to Vertex B, their priority values must satisfy $pri(A) < pri(B)$.

When we only consider one batch policy update consisting of multiple rule updates, the quality of the output is measured by the number of priority changes, i.e., the number of retained vertices whose priority values are changed. We define batch priority assignment problem as the problem to find the priority assignment with the minimum number of priority changes.

When we consider a sequence of batch policy updates, where upon each update the prioritizer does not know about the future updates, the quality of the output sequence is then measured by the total number of priority updates. We define it as online priority assignment problem, which is an online version of the previous optimization problem.

5.2 Batch Priority Assignment

We solve batch priority assignment problem optimally through dynamic programming. The key idea is to iteratively find the optimal priority assignment for a subset of the new flow table. The algorithm is detailed in **Algorithm 1**.

ALGORITHM 1: DYNAMIC PROGRAMMING ALGORITHM FOR BATCH PRIORITY ASSIGNMENT. n IS THE MAXIMUM PRIORITY VALUE. $updated(w, l)$ RETURNS 0 IF RULE w HAS PRIORITY VALUE l IN THE OLD FLOW TABLE, RETURNS $+\infty$ IF $l \le 0$ AND RETURNS 1 OTHERWISE.

```
  input : Annotated dependency graph $G =< V, E >$
  output: Priority assignment $f : V \rightarrow \{1, 2, ..., n\}$
1 $L \leftarrow$ a sorted list of $V$ in topological order
2 for $v \leftarrow 1$ to $|V|$ do
3     for $k \leftarrow 1$ to $n$ do
4         $PS[v][k] \leftarrow \sum_{(w,v)\in E} \min_{1\le l<k} PS[w][l] + updated(w, l)$
5 for $v \leftarrow |V|$ to 1 do
6     $ub \leftarrow \min_{(v,w)\in E} f[w]$
7     $f[v] \leftarrow \arg\min_{1\le k<ub} PS[v][k]$
```

The algorithm traverses the new flow table in the topological order regarding the dependency graph, so that when it visits a vertex, the optimal solution for all its parent vertices have been calculated. $PS[v][k]$ records the minimum number of priority changes of all v's ancestor vertices when v is assigned with priority value k. As it proceeds, the algorithm incrementally explores all possible priority assignments based on previous optimal solutions. Thus, this algorithm guarantees to output a global optimal priority assignment.

5.3 Online Priority Assignment

Assigning priority values entails a stochastic process requiring an online strategy: the previous priority assignment decision will become an "existing state" and affect priority assignment of the next policy update. A static optimization solution is impossible due to the uncertainty of the future updates. Instead, we opt for a heuristic approach based on the intuition that a more evenly scattered distribution of priority values reduces the chance of future priority changes.

We formulate the evenness of a priority distribution as the minimum priority gap, i.e., the smallest priority value difference between two adjacent rules. By maximizing the minimum priority gap, we achieve a more "balanced" priority value distribution.

We integrate the heuristic into the previous algorithm by using it to select the most balanced priority assignment among the huge amount of optimal assignments discovered by the previous algorithm. The algorithm is detailed in Algorithm 2. Specifically, $MG[v][k]$ records the evenness indicator of all optimal priority assignments of all v's ancestor vertices with v assigned to priority k.

5.4 Improve Algorithm Speed

Denoting the flow table size as m and the maximum priority number as n, the time complexity of Algorithm 1 and **Algorithm 2** is the state table size $O(mn)$ times the complexity of state transition function $O(n)$. Considering a typical m of thousands and n of 65536, it can take days to calculate an optimal assignment. Therefore, speeding up the algorithm is necessary.

Algorithm 2: Dynamic programming algorithm for online priority assignment.

```
input : Annotated dependency graph $G =< V, E >$
output: Priority assignment $f : V \rightarrow \{1, 2, ..., n\}$
1 $L \leftarrow$ a sorted list of $V$ in topological order
2 for $v \leftarrow 1$ to $|V|$ do
3     for $k \leftarrow 1$ to $n$ do
4         $PS[v][k] \leftarrow \sum_{(w,v)\in E} \min_{1\le l<k} PS[w][l] + updated(w, l)$
5         $MG[v][k] \leftarrow \min_{(w,v)\in E} \max_{l\in best(w,k)} min(MG[w][l], k-l)$,
          where $best(w,k) = \{l|DP[w][l] = \min_{1\le m<k} PS[w][m]\}$
6 for $v \leftarrow |V|$ to 1 do
7     $ub \leftarrow \min_{(v,w)\in E} f[w]$
8     $f[v] \leftarrow \arg\min_{1\le k<ub} MG[v][k]$
```

The key idea to speed up the algorithm is to compress the state table size. Intuitively, considering assigning priority for the highest priority rule in a sub-flow table, the minimum number of priority change ($PS[v][k]$) is always a step function of k, because the function value only reflects the combination of retained/changed state of the ancestor rules. Specifically, we can prove the step function only has no more than $3d$ stages, where d is the length of the longest path of the dependency graph. Therefore, instead of recording n PS values for each vertex, we only need to characterize the step function with less than $3d$ step points of k and the corresponding function values.

As a result, the time complexity of Algorithms 1 and 2 can be reduced to $O(md^2)$. In practice, the optimized algorithms typically calculate the optimal assignments within a few hundred milliseconds for flow tables with thousands of rules.

6 Back-End Optimizer

The DAG and DAG updates generated by RuleTris front-end are exploited by RuleTris back-end to conduct optimization to TCAM updates. RuleTris 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.

6.1 Update Scheduler

The update scheduler exploits the DAG to optimize the rule update process in TCAM. 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 3** 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 **Fig. 8**). Finally, the new rule is inserted by moving every rule in the moving chain one slot downstream. The optimality proof of Algorithm 3 is provided in the Appendix.

For example in Fig. 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 is the same, a final update decision is picked on a random basis.

6.2 Redundancy Eliminator

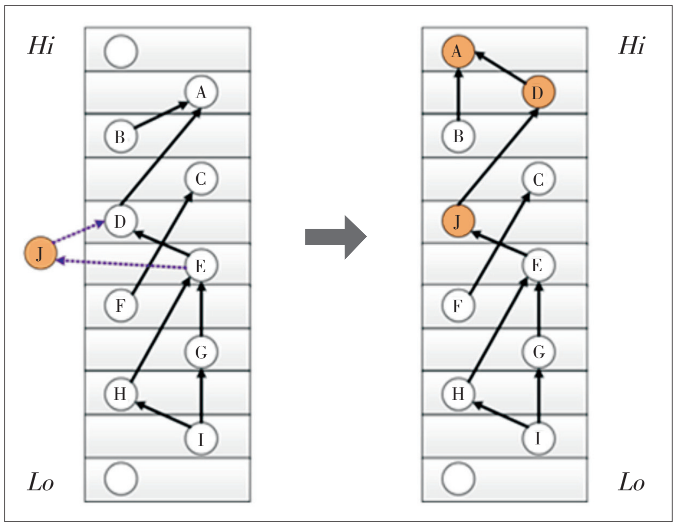
The redundancy eliminator uses the DAG to remove redundant rules. Specifically, we observe two types of redundancy in flow tables:

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.

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 an 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

ALGORITHM 3: SHORTEST MOVING CHAIN SEARCH.

```
Input  : Rule DAG $< V, E >$, TCAM layout $f_r : Addr. \to V$, 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 $\{r \mid < r_{curr}, r > \in E\}$ 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 $\{r \mid < r, r_{curr} > \in E\}$ 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 $d_{pre}$
```



▲Figure 8. Example of TCAM move optimization.

rule and should be removed.

6.3 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.

7 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 ONetSwitch firmware 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 use the original ONetSwitch firmware with full OpenFlow v1.3 support.

8 Evaluation

8.1 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 as 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 [22]. The front-end compilation and the back-end emulation are done on a Linux workstation with 4 cores at 2.8 GHz and 8 GB 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 struc-

ture. 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 [23] 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/User Datagram Protocol (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/swapout operations is randomly generated to mimic the cache swapping behavior.

c) Metrics

In Figs. 9, 10, and 11 the bars show the median, and the error bars show the 10th and 90th percentiles.

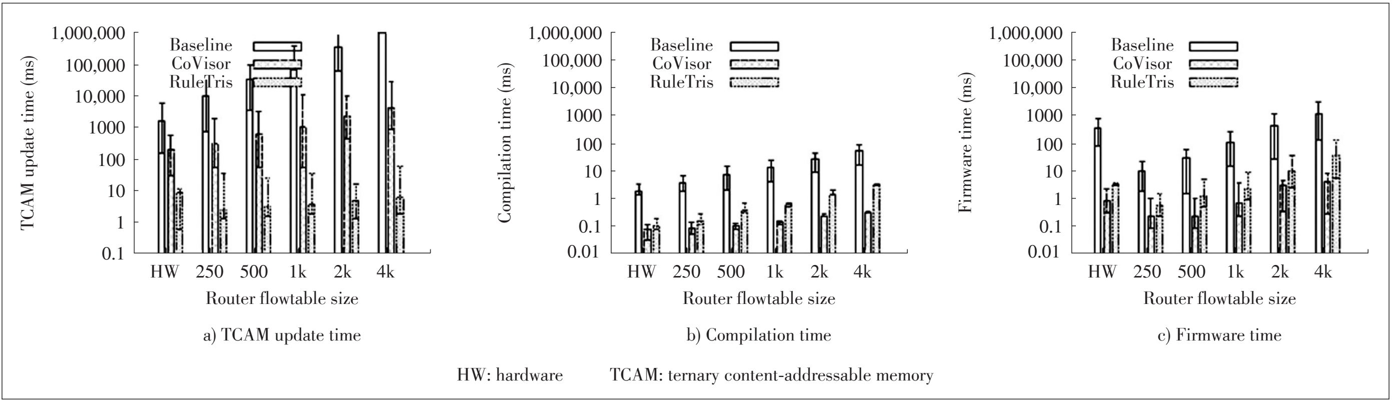
- Compilation time. It is the computation time for compiling the rule update in the front-end.
- Firmware time. It is the computation time for calculating the update schedule from a priority-based or dependency graph-based rule update in the switch firmware. In hardware experiments, this time is measured on the 800 MHz 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. It is 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 (0.6 ms) to estimate the TCAM update time in emulation experiments.

8.2 Experimental Results

Fig. 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 TCAM update time, compilation time and firmware time are shown in Figs. 9a, 9b and 9c 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 fastest and a fairly constant latency in TCAM updates. 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.

Fig. 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 removed from and one rule inserted 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.



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

Fig. 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 Figs. 11a and 11b respectively. As expected, RuleTris's DAG based updates show 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.

9 Discussion

1) Multiple Tables

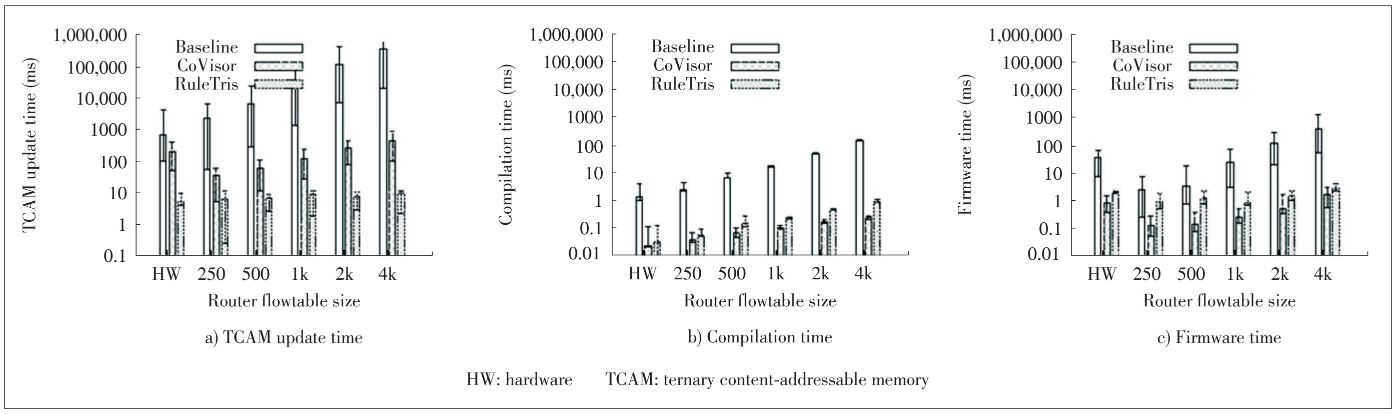
RuleTris currently optimizes updates to a single flow table. Switches typically have multiple tables. Depending on the order of execution of the tables, we can further minimize the rule updates. For example, if we have two TCAM tables in a pipeline, the dependencies between the two modules in a sequential composition can be decoupled by placing the first one in the first TCAM and the second module in the second TCAM. Similarly, if we have two TCAM tables that operate in parallel and the actions are both applied, we can decouple the dependencies of the two modules in a parallel composition. 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.

2) Hardware Specific Optimizations

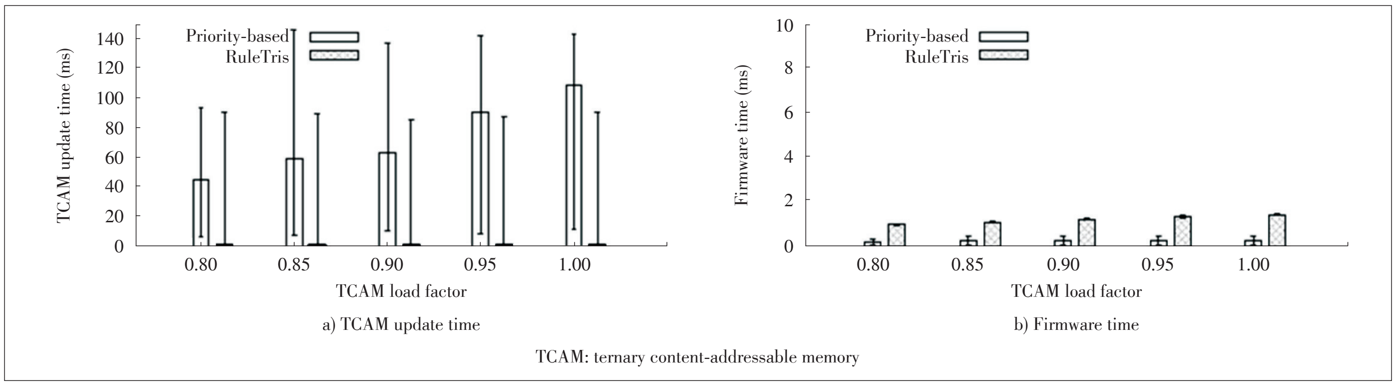
Tango [24] and Mazu [18] have shown that different switches can have very different latency behavior depending on the order of rule updates. For example, given two ordering of a batch of rules, one is in increasing priority and the other in decreasing priority. One switch has a much lower latency for the first order. Techniques [18], [24] proposed to exploit hardware behavior can be usefully combined with RuleTris.

3) Minimal Network Update

RuleTris considers per switch flow table updates independently. Coordination among flow tables of several switches can



▲**Figure 10. Rule update overhead of L3-L4 network address translation (NAT) > L3 router.**



▲**Figure 11. Rule update overhead of single rule swaps with CacheFlow. Results are from hardware experiments.**

be combined with RuleTris to further reduce the number of updates [5], [25]–[28].

For future work, we would like to release the source code of RuleTris and build RuleTris into controller platforms such as OpenDaylight and Open Network Operating System (ONOS), and hypervisors such as OpenVritex. We would also like to exploit the benefits of multiple flow tables and the gains across switches.

10 Conclusions

To enable effective modular programming in software defined networks, it is crucial that modular composition be optimized end-to-end for both compilation time and switch update time and flow table size. 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.

Appendix

We first introduce the definition of a valid entry moving chain. We define an entry moving chain as valid if after inserting the new rule by moving entries along the entry moving chain, all the entries still satisfy the dependency constraints indicated by the DAG.

Now we prove Algorithm 3 generates one of the shortest valid entry moving chains.

Theorem 1. Algorithm 3 generates a valid entry moving chain.

Proof: There are two types of entry moves in the generated entry moving chain.

First, the new rule r_{insert} is written into the location of an existing rule (the loop between Line 6 and Line 8). Considering the range of the loop variable, the possible destination location of r_{insert} is from the location of r_{insert}'s highest predecessor r_{pre} and the location of r_{insert}'s lowest successor r_{succ}. If the destination location is between r_{pre} and r_{succ} exclusively, it is obviously that the destination location of r_{insert} is higher than all its predecessors and lower than all its successors, thus the dependency holds. If the destination location is at r_{pre} (r_{succ}), the following moving chain searching code at Line 26 (Line 18) determines that r_{pre} (r_{succ}) is moved to the a lower (higher) location. Therefore the dependency holds.

Second, some existing rules are moved from the previous location to a new location (the two loops between Line 9 to Line 23). Without loss of generality, we consider the downstream search loop (Line 9 to Line 15), which searches for the shortest downstream moving chain. Specifically, Line 15 determines that the possible destination location of an existing rule r_{curr} is between $r_{curr}.addr+1$ and r_{curr}'s lowest successor's location. Therefore, the new location of r_{curr} is still lower than all its successors.

Theorem 2. Given the input rule DAG is minimum, no other valid entry moving chain has fewer entries than the entry moving chain generated by Algorithm 3.

Proof: Without loss of generality, we still only consider the downstream search. We prove the theorem by induction on the loop variable i of the loop from Line 9 to Line 15.

Base case: When $i=r_{pre}.addr+1$, the length of the moving chain is one (set by at loop from Line 6 to Line 8), and it is the minimum possible length of a valid entry moving chain.

Induction: Assuming for all $i=r_{pre}.addr+1$ to i_{curr-1}, the shortest entry moving chains are known, i.e., $f_r(i).move$ and $f_r(i).prev$ store the correct length of its shortest entry moving chain.

Consider $i=i_{curr}$. Assuming the previous entry on the shortest entry moving chain is i_{last}, the index of the lowest successor of $f_r(i_{last})$ must be larger or equal to i_{curr}, because otherwise moving $f_r(i_{last})$ to i_{curr} would introduce a DAG edge inversion between $f_r(i_{last})$ and its lowest successor. Since the DAG is minimum, a DAG edge inversion is necessarily a dependency violation.

The loop between Line 15 to Line 18 guarantees that if $f_r(i_{last})$ is larger or equal to i_{curr}, $f_r(i_{last})$ must have been considered to move to i_{curr}, therefore we have

$$f_r(i_{curr}).move \leqslant f_r(i_{last}).move+1. \tag{1}$$

On the other hand, for all i'_{last} that has $f_r(i'_{last}).move=f_r(i_{last}).move-1$, the index of the lowest successor of $f_r(i'_{last})$ must be smaller than i_{curr}, because otherwise moving $f_r(i'_{last})$ to i_{curr} would be valid and $f_r(i_{last})$ would not be the previous entry of $f_r(i_{last})$ on the shortest entry moving chain, which contradicts with the assumption. The loop between Line 15 to Line 18 guarantees $f_r(i'_{last})$ will not be considered to move to i_{curr}, therefore we have

$$f_r(i_{curr}).move > f_r(i'_{last}).move+1=f_r(i_{last}). \tag{2}$$

Since all the values are integers, we can combine 1 with 3 and have

$$f_r(i_{curr}).move=f_r(i_{last}).move+1, \tag{3}$$

which is the correct length of $f_r(i_{curr})$'s shortest moving chain.

References

[1] *Requirements of an MPLS Transport Profile*, IETF RFC 5654, 2009.

[2] M. Al - Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, USA, 2010, pp. 19–19.

[3] ONF. (2013, Oct. 8). *Solution brief: SDN security considerations in the data center* [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-security-data-center.pdf

[4] M. Kuzniar, P. Perešíni, and D. Kostic, "What you need to know about SDN flow tables," in *International Conference on Passive and Active Measurement*, New York, USA, 2015, pp. 347–359. doi: 10.1007/978-3-319-15509-8_26.

[5] X. Jin, H. H. Liu, R. Gandhi, et al., "Dynamic scheduling of network updates," in *ACM Conference on SIGCOMM*, Chicago, USA, 2014, pp. 539–550. doi: 10.1145/2619239.2626307.
[6] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: a compositional hypervisor for software-defined networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI' 15)*, Oakland, USA, 2015, pp. 87–101.
[7] X. T. Wen, C. X. Diao, X. Zhao, et al., "Compiling minimum incremental update for modular SDN languages," in *Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Chicago, USA, 2014. doi: 10.1145/2620728.2620733.
[8] J. Van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560–571, May 2003. doi: 10.1109/JSAC.2003.810527.
[9] T. Mishra and S. Sahni, "DUO-dual TCAM architecture for routing tables with incremental update," in *IEEE International Symposium on Computers and Communications (ISCC)*, Riccione, Italy, 2010, pp. 503–508. doi: 10.1109/ISCC.2010.5546713.
[10] H. Y. Song and J. Turner, "Fast filter updates for packet classification using TCAM," in *IEEE GLOBECOM*, San Francisco, USA, 2006. doi: 10.1109/GLOCOM.2006.342
[11] D. Shah and P. Gupta, "Fast updating algorithms for TCAMs," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan. 2001. doi: 10.1109/40.903060.
[12] A. Voellmy, J. Wang *et al.*, "Maple: simplifying SDN programming using algorithmic policies," in *ACM SIGCOMM*, Hong Kong, China, 2013, pp. 87–98. doi: 10.1145/2534169.2486030.
[13] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Chicago, USA, 2014, pp. 175–180. doi: 10.1145/2620728.2620734.
[14] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. (2013). *Composing software defined networks* [Online]. Available: http://frenetic-lang.org/publications/composing-nsdi13.pdf
[15] C. J. Anderson, N. Foster, A. Guha, *et al.*, "NetKAT: semantic foundations for networks," in *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 14)*, San Diego, USA, 2014. doi: 10.1145/2535838.2535862.
[16] C. C. Hu, J. Yang, H. B. Zhao, and J. H. Lu. (2014). Design of all programmable innovation platform for software defined networking [Online]. Available: https://www.usenix.org/system/files/conference/ons2014/ons2014 - paper - hu_chengchen.pdf
[17] ONetSwitch. (2018). *ONetSwitch45* [Online]. Available: http://onetswitch.org/hardware45.html
[18] K. He, J. Khalid, S. Das, et al., "Mazu: taming latency in software defined networks," University of Wisconsin-Madison, Tech. Rep., 2014.
[19] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006. doi: 10.1109/JSSC.2005.864128.
[20] N. Foster, R. Harrison, M. J. Freedman, et al., "Frenetic: a network programming language," in *16th ACM SIGPLAN international conference on Functional programming*, Tokyo, Japan, 2011, pp. 279–291. doi: 10.1145/2034773.2034812.
[21] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, USA, 2012, pp. 217–230. doi: 10.1145/2103656.2103685.
[22] Ryu SDN Framework Community. (2015, Jan. 29). *Ryu OpenFlow controller* [Online]. Available: https://osrg.github.io/ryu/index.html
[23] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, Jun. 2007. doi: 10.1109/TNET.2007.893156.
[24] A. Lazaris, D. Tahara *et al.*, "Tango: simplifying SDN programming with automatic switch behavior inference, abstraction, and optimization," in *ACM International on Conference on Emerging Networking Experiments and Technologies (CoNext)*, Sydney, Australia, 2014, pp. 199–211. doi: 10.1145/2674005.2675011.
[25] C.-Y. Hong, S. Kandula, R. Mahajan, et al., "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM*, Hong Kong, China, 2013, pp. 15–26. doi: 10.1145/2486001.2486012.
[26] H. H. Liu, X. Wu, M. Zhang, et al., "zUpdate: updating data center networks with zero loss," *ACM SIGCOMM*, Hong Kong, China, 2013. doi: 10.1145/2534169.2486005.
[27] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, Helsinki, Finland, 2012. doi: 10.1145/2342356.2342427.
[28] N. P. Katta, J. Rexford *et al.*, "Incremental consistent updates," in *Second Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Hong Kong, China, 2013, pp. 49–54. doi: 10.1145/2491185.2491191.

Manuscript received: 2018-07-17

Biographies

CHEN Yan (ychen@northwestern.edu) received the Ph.D. degree in computer science from the University of California at Berkeley, USA, in 2003. He is currently a professor with the Department of Electrical Engineering and Computer Science, Northwestern University, USA and a distinguished professor with the College of Computer Science and Technology, Zhejiang University, China. Based on Google Scholar, his papers have been cited over 10,000 times and his h-index is 49. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. He received the Department of Energy Early CAREER Award in 2005, the Department of Defense Young Investigator Award in 2007, the Best Paper nomination in ACM SIGCOMM 2010, and the Most Influential Paper Award in ASPLOS 2018.

WEN Xitao (xitao.wen@gmail.com) received the B.S. degree in computer science from Peking University, China, in 2010, and the Ph.D. degree in computer science from Northwestern University, USA, in 2016. His research interests span the area of networking and security in networked systems, with a current focus on software-defined network security and data center networks.

LENG Xue (lengxue_2015@outlook.com) received the B.S. degree in computer science and technology from Harbin Engineering University, China, in 2015. She is currently pursuing the Ph.D. degree major in computer science and technology with Zhejiang University, China. Her research interests are software-defined networking (SDN), network function virtualization (NFV), microservice and 5G protocol verification. She is a student member of the IEEE and CCF.

YANG Bo (ybo2013@zju.edu.cn) received the B.S. degree in information security from the Huazhong University of Science and Technology, China, in 2013, and the M.S. degree in computer science from Zhejiang University, China, in 2016. He is currently a software engineer with Microsoft, Shanghai, China. His research interests include software-defined network and network security.

Li Erran Li (lierranli@gmail.com) received the Ph.D. degree in computer science from Cornell University, USA. He was a researcher with Bell Labs. He is currently with Uber and also an adjunct professor with the Computer Science Department, Columbia University, USA. His research interests are in machine learning algorithms, artificial intelligence, and systems and wireless networking. He is an ACM Distinguished Scientist. He was an associate editor of the IEEE Transactions on Networking and the IEEE Transactions on Mobile Computing. He co-founded several workshops in the areas of machine learning for intelligent transportation systems, big data, software defined networking, cellular networks, mobile computing, and security.

ZHENG Peng (zeepean@gmail.com) received the B.S. degree in information security from Northwestern Polytechnical University, Xi'an, China, in 2015. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Xi'an Jiaotong University, China. He was a visiting research fellow at Duke University, USA from July to August 2017 and Brown University from July to October 2018, respectively. He has authored papers in CoNEXT, ICDCS, ICNP, etc. His research interests span the area of computer networking and systems, with a focus on the programmable network and software-defined networking.

HU Chengchen (chengchenhu@gmail.com) received the B.S. degree from the Department of Automation, North-Western Polytechnical University, China, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2003 and 2008, respectively. He worked as an assistant research professor with Tsinghua University from July 2008 to December 2010. After that, he joined the Department of Computer Science and Technology, Xi'an Jiaotong University, China, where he is currently a full professor. His main research interests include computer networking systems and network measurement and monitoring.