Characterizing Rule Compression Mechanisms in Software-defined Networks

Curtis Yu[1], Cristian Lumezanu[2], Harsha V. Madhyastha[3], Guofei Jiang[2]

[1] University of California, Riverside
[2] NEC Labs America
[3] University of Michigan

Abstract. Software-defined networking (SDN) separates the network policy specification from its configuration and gives applications control over the forwarding rules that route traffic. On large networks that host several applications, the number of rules that network switches must handle can easily exceed tens of thousands. Most switches cannot handle rules of this volume because the complex rule matching in SDN (*e.g.*, wildcards, diverse match fields) requires switches to store rules on TCAM, which is expensive and limited in size.
We perform a measurement study using two real-world network traffic traces to understand the effectiveness and side-effects of manual and automatic rule compression techniques. Our results show that not using any rule management mechanism is likely to result in a rule set that does not fit on current OpenFlow switches. Using rule expiration timeouts reduces the configuration footprint on a switch without affecting rule semantics but at the expense of up to 40% increase in control channel overhead. Other manual (*e.g.*, wildcards, limiting match fields) or automatic (*e.g.*, combining similar rules) mechanisms introduce negligible overhead but change the original configuration and may misdirect less than 1% of the flows. Our work uncovers trade-offs critical to both operators and programmers writing network policies that must satisfy both infrastructure and application constraints.

1 Introduction

Software-defined networking (SDN) enables flexible and expressive network management by separating the policy specification from configuration. Applications and operators work with abstract network views [19] and specify policies using an API. A centralized controller program translates the high-level policies into low-level configurations—expressed as forwarding rules—and installs them into the switch memory using a specialized protocol, such as OpenFlow [16].

To maintain network performance, the set of forwarding rules installed at a switch must fit into the switch's memory. Two factors complicate this. First, as more applications adopt SDN, the number of rules required to express their policies on every switch grows, similar to how BGP tables have grown with the spread of the Internet. Researchers have observed that an average top-of-rack (ToR) switch would have to hold around 78K rules with the default expiration timeout [5,12]. Second, switches store wildcard rules in TCAM, which is expensive and limited in size. Most programmable switches can hold only a few thousand wildcard-based rules.

There are two general approaches to ensure that application policies do not result in too many rules: *compression* and *caching*. Network control programs can reduce the number of rules manually (by relying on programmers to employ OpenFlow constructs such as rule expiration timeouts or wildcards [27,5]) or automatically (by eliminating redundant rules or combining rules with related patterns). Compression may limit the expressivity of the configuration as it changes the original rule space. In addition, when rules are generated in response to traffic, it is difficult to predict how many rules we need *a priori* to tune the compression accordingly. Another approach is to cache the most popular rules in TCAM and rely on additional (software) switches or the controller to manage traffic not matching the cached rules [13]. This preserves the original configuration but may introduce additional devices and delay in the data plane of packets matching less popular rules.

In this paper, we use two sets of real world network traffic data to study the *effectiveness* and *side-effects* of manual and automatic rule compression. We seek to answer the following questions: *should SDN rely on programmers to employ mechanisms that reduce the number of rules installed on switches and if so, what are the most effective such mechanisms?* or *can SDN benefit from an automated rule reduction system that sits between the controller and switches and optimizes how rules are installed on switches?* Our work explores trade-offs critical to both operators and programmers writing network policies that must satisfy both infrastructure and application constraints.

First, we show how existing mechanisms that programmers and applications employ, such as reducing rule expiration timeout, using wildcards, or limiting the match fields, manage the rules on a switch (Section 4). Lowering rule timeouts can reduce the number of rules by 41–79%, as compared to the default operation, but at the expense of increasing the utilization on the constrained controller-to-switch channel by up to 40%. Even such high compression rates may be insufficient for most OpenFlow switches on the market. Using wildcards or limiting the match fields can further improve the configuration footprint but also limits the expressivity of the configuration as the original rule semantics change.

Second, we show that automatic rule compression can benefit SDN. We introduce and evaluate a simple mechanism that encodes rules using binary trees to identify and combine similar rules. (Section 5). This reduces the configuration size on a switch by as much as 62% compared to normal operation and at little change in network overhead. However, such benefit comes at a cost: aggressive automatic rule compression can also result in some flows ($<$1%) being misdirected.

2 Motivation

In this section, we discuss how programmable switches store rules and implement rule matching. We also review related research work and potential solutions for reducing the number of rules. To keep the discussion simple, we consider OpenFlow as the de facto protocol for installing and managing switch configurations.

2.1 Rules and memory

A network's configuration consists of the forwarding rules installed at the switches. Every rule consists of a bit string (with 0, 1, and * as characters) that specifies which packets match the rule, an action (to be performed by the switch on matched packets), and a set of counters (which collect statistics). Possible actions include "forward to physical port", "forward to controller", "drop", etc. Each rule has two expiration timeouts: a soft one, counted from the time of the last packet that matched the rule, and a hard one, from the time when the rule was installed.

Switch	**Max # rules**	**Source**
NEC PF5820	750	[1]
HP ProCurve 5406zl	1500	[5]
Pronto 3290	4000	[2]
HP 3800	10k (routing)	[10]
NEC PF5240	64k-160k	[1]
IBM G8264	97k	[11]

Table 1: Several OpenFlow switches specify the maximum number of forwarding rules that they store. Each rule can contain any subset of the 12 fields specified in the OpenFlow v1.0 specification [22], which is used by most switches on the market. The HP 3800's fact sheet specifies the maximum number of routing, rather than OpenFlow, entries; a routing entry can be considered an OpenFlow entry with matches only on layer 3 fields.

Implementation details of how rules are stored and matched is left to the discretion of each switch vendor [24]. A common approach is for switches to store wildcard rules in TCAM and exact match rules in SRAM. TCAM is fast and can support wildcards efficiently. However, since it is also expensive and power hungry, its size on switches is limited. On the other hand, SRAM is cheaper and is available in higher capacity, but has a higher lookup latency because it is often off-chip and uses search structures (*e.g.*, hash tables and tries) to locate entries.

Switch vendors do not advertise the details of their OpenFlow implementation. In addition, the number of OpenFlow rules that a switch can store in hardware is not always fixed and depends on how rules are formed (*e.g.*, whether they have wildcards, what fields they match on). We studied the public datasheets for six popular OpenFlow switches and compiled their published OpenFlow table limits in Table 1. Unless otherwise noted, the numbers correspond to 12-tuple OpenFlow rules. Independent measurements and personal communication with vendors indicate that the values are representative for current OpenFlow switches [24,3]. Prior work [5,12] has observed that a typical ToR data center switch may store roughly 78K rules, an order of magnitude larger than most switches in the table. Although architectural and algorithmic advances in switch design may extend the memory limits further (*e.g.*, by using memory other than TCAM or by making software lookups faster), reducing the configuration size to begin with is still essential to preserve flexibility and minimize the cost of lookups.

2.2 Managing configuration size

There are two types of solutions to manage configuration size: architectural-based and software-based. Architectural-based solutions seek to optimize the performance of a switch through various architectural design changes [2], but are slow to develop and integrate. Software-based methods seek to reduce the size of the configuration that can be stored on current architectures. We focus on software-based configuration size management and discuss the two main approaches: compressing the rule set and caching the more popular rules. In this paper, we study compression-based techniques.

Compression Compression-based mechanisms are automatic (*i.e.*, without programmer involvement) or manual (*i.e.*, require actions from the programmer).

Manual. Personal communication with SDN operators and previous work [5,27] indicate several OpenFlow-based mechanisms to reduce the flow table size on a switch. These methods limit the number of rules by *having existing rules cover more traffic* [5] (*e.g.*, using wildcards rather than exact matches, using fewer match fields) or *cover the same traffic for shorter periods of time* (*e.g.*, setting smaller rule expiration timeouts). However, this also results in a less expressive configuration because it reduces the ability to implement complex policies, such as multipathing [21]. Furthermore, wildcards and longer timeouts reduce visibility into the network as they increase the coarseness of the statistics that switches gather about flows.

Automatic. Rule management has been studied in the context of IP routing table compaction [25], with the goal of restricting the usage of TCAM [15,23]. While some of these methods (e.g., [15]) use binary trees to identify similar rules (like the approach we present later in Section 5), existing methods work on a "single IP to out port" action and are not easily applicable to OpenFlow rules, which may have as many as 12 different match fields to be aggregated at once. The TCAM Razor approach uses decision trees and multi-dimensional topological transformations to efficiently compress packet-classification rules [17,14], but cannot easily adapt to incremental rule changes. To the best of our knowledge, none of these methods have been implemented in an OpenFlow-based network.

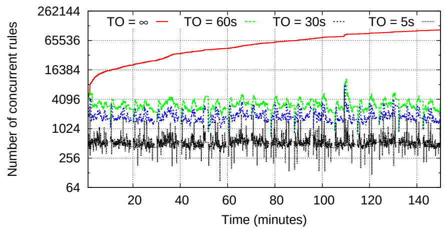
Policy composition and arbitration frameworks such as Frenetic [7], NetCore [18], and PANE [8] manage application policies to ensure that there are no conflicting or overlapping rules. vCRIB [20] intelligently places rules on different OpenFlow switches while being aware of the resources that the rules utilize. Although these systems can optimize the rules they place on switches (*e.g.*, by eliminating redundancies), their focus is on managing the policies installed across the network, rather than on reducing the configuration size on any single switch.

Caching Rather than compressing the rule set, Katta *et al.* propose to keep only the more popular rules in TCAM and use additional (software) switches or the network controller to manage the traffic that does not match on the cached rules [13]. This approach preserves the semantics of the original rule space at the expense of additional devices or delay on the data path of a subset of the traffic.

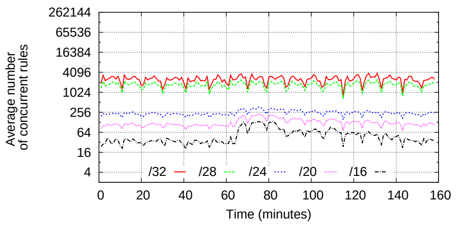
3 Method and data

We use two traces of real-world network traffic to characterize the effectiveness of manual and automatic rule compression techniques in reducing the flow table size.

Data. We use a packet trace from a campus network and a flow-level trace from a nation-wide research network. Our goal is to assess the *potential* of rule compression



(a)



(b)

Fig. 1: (a) Maximum number of concurrent rules as we vary the timeout after which rules expire and (b) average number of rules over time as we vary the IP prefix size from /32 to /16.

mechanisms *when regular network traffic traverses OpenFlow devices*. Thus, our traces are not collected from OpenFlow-based networks, whose traffic may already be adapted to the programmabile nature of the network. The first dataset, *Campus*, was collected by Benson *et al.* [4] at an edge switch of a large US campus network in Jan 2010 and contains 115K flows over two hours. The second dataset, *Abilene*, contains 1% sampled Netflow data from the Internet2 network, collected at the Washington, DC router in Feb 2013. The trace contains around 12M flows over three hours. For anonymity, the IP addresses in the *Abilene* trace have their last 11 bits zeroed out. These two datasets are typical for two important OpenFlow switch usage scenarios: at the edge and at the core of a network.

Rule generation. Since neither of the two networks where the traces were collected is OpenFlow-enabled, we simulate the operation of an OpenFlow network to determine the set of rules that would be installed to handle the traffic. We first identify all five-tuple flows (src IP, dst IP, src port, dst port, protocol) in each dataset and assume that each flow must be handled by one rule (*i.e.*, with no wildcards). We create matches on five fields rather than all 12 supported by existing OpenFlow switches because these are the fields for which our traces include information. We assume a switch with a single flow table, conforming to OpenFlow v1.0, which is implemented on most switches on the market.

As the data sets do not have any information about the actual out port number used for every flow, we use the following heuristics to determine the action of each rule. Since the *Abilene* dataset contains next-hop IP information, we associate every next-hop IP with a unique out port. For the *Campus* data set, we simulate a 24-port switch, where every flow is assigned to an out port based on its destination IP prefix. We assume a reactive OpenFlow deployment (*i.e.*, the installation of the rule corresponding to a flow is triggered by the first packet in the flow and its removal by the timeouts), as it offers a dynamic model for rule management and a worst case scenario for evaluation (because it maximizes the total number of rules that are generated).

Evaluation metrics. To measure the effectiveness of rule reduction techniques, we use the maximum value across time of the *total number of rules installed on the switch* at any moment in time. To measure the side effects of reducing the number of rules, we measure the *rate of controller-to-switch operations* (to estimate overhead).

	Campus		*Abilene*	
	# rules	ops/sec	# rules	ops/sec
no mgmt.	**115K**	**46**	**12M**	**1255**
60s timeout	**11K**	**176**	**100.5K**	**2800**
Timeouts (§4.2)				
- 30s	7,982 (-27%)	200 (+14%)	53K (-47%)	2,914 (+1.2%)
- 10s	6,757 (-39%)	233 (+32%)	29K (-71%)	3,214 (+12%)
- 5s	6,509 (-41%)	247 (+40%)	21K (-79%)	3,631 (+26%)
Match fields (§4.3)				
- dest-only	7,052 (-36%)	73 (-59%)	75K (-26%)	1,949 (-32%)
- IP-only	4,460 (-59%)	125 (-29%)	53K (-47%)	1,215 (-58%)
Wildcard (IP granularity) (§4.4)				
- \24	8479 (-23%)	69 (-61%)	-	-
- \16	8225 (-25%)	66 (-63%)	100K (-0%)	2,784 (-3%)
- \8	8218 (-25%)	66 (-63%)	99K (-1.5%)	2,752 (-4%)
IP-only, 60s	**4,460**	**125**	**53K**	**1,215**
Simple aggregation (§5.1)				
T = 100%	3,568 (-20%)	121 (-3%)	46K (-13%)	1,277 (+5%)
Aggressive aggregation (§5.2)				
T = 25%	1,695 (-62%)	69 (-45%)	40K (-24%)	1,189 (-2%)
T = 50%	2,676 (-40%)	85 (-32%)	43K (-19%)	1,234 (-1%)
T = 75%	3,122 (-30%)	106 (-15%)	45K (-15%)	1,265 (0%)

Table 2: Comparison of various rule management methods. For each method, we show for both datasets the maximum number of concurrent rules and the 95^{th} percentile value (across minutes) of operations per second. Percentages for number of rules and ops/sec are in comparison to the default OpenFlow operation of using a 60s timeout (for the manual techniques) and to the IP-only rules with 60s timeout (for the automatic aggregation).

4 Manual Rule Management

In this section, we study manual solutions for reducing the number of rules on an OpenFlow switch. These are solutions that programmers must proactively use in their code. We derive them from personal communication with SDN operators and previous work [5,27]. These mechanisms limit *the time a rule stays on the switch* (through rule expiration timeouts), *the space occupied by a rule on the switch* (by reducing the number of fields to match on), or *the total number of rules* (by using wildcards).

4.1 Not managing rules

Figure 1(a) shows the number of concurrent rules that would have to be held on an OpenFlow switch that forwards the flows in the *Campus* dataset. We assume rules do not expire (TO = ∞) and contain exact matches on all five fields mentioned in Section 3. Since rules never expire, their number is continually increasing as new flows arrive, reaching a maximum of 115,323 rules at the end of the trace. We repeat the experiment for the *Abilene* data and find that it generates more than 12M rules. Recall however that the *Abilene* IPs have their last 11 bits zeroed, therefore the rules are essentially wildcard rules; the number of exact match rules will be much higher. These numbers exceed the maximum number of flows supported by all but one of the OpenFlow switches described in Table 1.

4.2 Timeouts

We vary the soft timeout for each rule from 5s to 60s (the default timeout value in OpenFlow). Rules with short timeouts are expunged sooner and may need to be reinstalled if there are subsequent packets matching the rule. Large timeouts keep the rule in memory longer and are suited for long flows with lower packet arrival rates. Figure 1(a) and Table 2 show that, as the soft timeout becomes smaller, the number of concurrent rules decreases and the rate of operations increases. Current switches typically handle around 275 operations (*i.e.*, rule installations or deletions) a second [5] and could support the 95^{th} percentile operation load in the *Campus* dataset but not in the *Abilene* trace.

4.3 Match fields

Having fewer match fields should decrease the memory footprint of an OpenFlow rule. We consider two smaller matches: on IP-only (source and destination, no ports) and on destination-only (destination IP and port, no source). Table 2 shows that both destination-only and IP-only matches lower the number of concurrent rules by at least 26%, as compared to 5-tuple rules with 60s timeout. Though these rule savings are significant, the maximum number of concurrent rules with the *Abilene* trace is still quite high compared to the memory capacity of three of the OpenFlow switches in Table 1. While fewer rules result in a lower rate of operations on the switch, since the flow arrival is not uniform, the 95^{th} percentile rate of operations per second in the *Abilene* trace is over 4x higher than the threshold of 275.

4.4 Wildcards

Wildcard-based rules cover a larger part of the flow-space and thus, fewer rules are necessary. However, they limit 1) the expressivity of the configuration because they cannot perform fine-grained matching (*e.g.*, for multipathing [21]), and 2) the application's visibility into the network because the controller cannot request statistics on the individual flows that match the rule.

To evaluate the effect of wildcards on the flow table size, we consider the original 5-tuple rules, as well as the destination-only and IP-only rules. For each rule, we introduce wildcards in the rightmost bits of IP addresses, effectively reducing them to prefixes. Figure 1(b) and Table 2 show that the average number of rules over each minute decreases as we vary the IP prefix size from /32 to /16. The savings (23% in the *Campus* data) come at the expense of more policy violations (30% of packets are forwarded differently). Combining wildcards with fewer match fields further reduces the number of rules, but not always sufficiently enough to fit into the memory of all switches in Table 1. The reduction in number of rules is lesser in the *Abilene* data because it includes

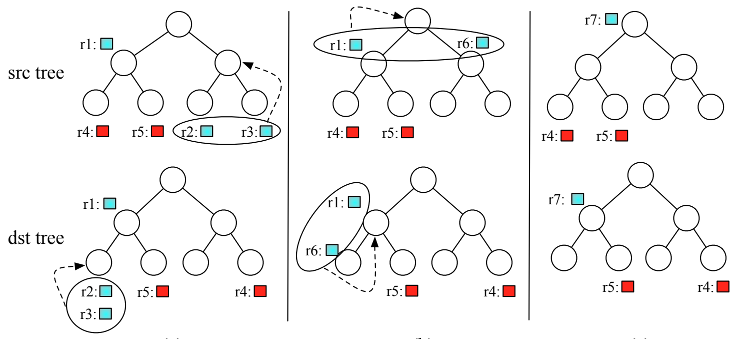


Fig. 2: Simple binary tree aggregation. For simplicity, we represent the subtrees corresponding to the last two bits of source and destination IPs. See Figure 3 for example rules mapped on these su'

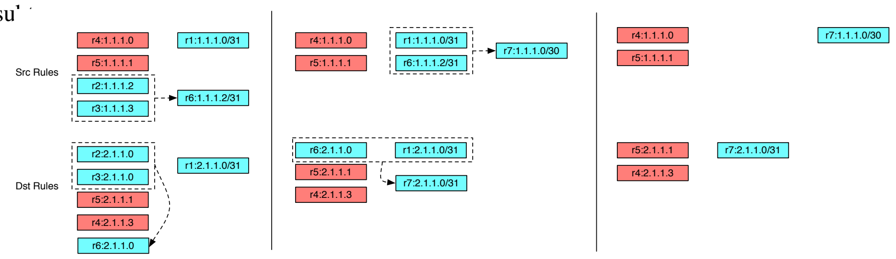


Fig. 3: Examples of rules mapping to the subtrees in Figure 2.

only /21 addresses. As with limiting the match fields, using wildcards reduces the expressivity of the installed configuration and our ability to retrieve information about the original rule set (*e.g.*, counters) as the rule semantics change.

4.5 Summary

The most consistently effective way to reduce the number of rules is by lowering rule expiration timeouts. Although it introduces a large network overhead because of the increased control channel operation rate, it preserves the original rule semantics and the controller's ability to query the counters of the original rules. Other approaches limit the control channel overhead at the expense of changing the original rule semantics.

No manual rule compression method is a panacea: as Table 2 shows, even in the best case compression scenario, the number of rules for *Abilene* cannot fit on half of the switches in Table 1. In reality, the type of traffic and the goal of network operators, in addition to rule compression algorithms, play a large role in determining how to fit the configuration on switches.

5 Automatic Rule Management

We now consider the scenario where the OpenFlow controller uses an automatic mechanism to reduce the number of rules. To the best of our knowledge there is no existing mechanism for rule space compression for SDN controllers. Existing rule compression approaches focus on IP routing table compaction [15,23] or minimizing packet classifiers in TCAM [17,6]. They use binary trees or decision trees to identify redundant and similar rules and focus on simple IP-based rules or on how to optimize ranges that cannot be stored as a simple prefix. Their applicability to OpenFlow is not clear yet as OpenFlow rules are more complex (up to 12 matching fields) [14]. Furthermore, IP-based rule management techniques cannot easily adapt to incremental rule changes.

To understand the potential of automatic rule compression, we propose a simple approach, based on the work of Liu [15], that uses binary trees to identify and aggregate related rules. In doing so, our goal is to provide a simple compression baseline. We do not seek to either introduce a novel OpenFlow table compaction method or to fully replicate and compare with previous rule aggregation methods built for IP-based rules. Evaluating these approaches within the scope of OpenFlow is subject to future work.

5.1 Simple aggregation

To reduce the memory footprint of the configuration installed on a switch, we automatically aggregate *similar* rules into a single rule. A network controller can accomplish this by intercepting all OpenFlow control messages and storing the state of all switches in-memory. On a rule install to a switch, the controller adds the rule to its in-memory state for the switch and checks for aggregation. If aggregation is not possible, the controller simply installs the rule into the switch. Otherwise, it sends an aggregated rule and deletes all rules that are covered by it. Similarly, on a rule removal, the controller checks to see if it is part of any aggregated ruleset and appropriately reinserts rules as necessary.

To build a proof of concept implementation of rule reduction and demonstrate its effectiveness, we use binary trees [26] to store and aggregate rules on a particular switch. Because we use binary trees, we are limited to only IP-based rules. We are currently exploring other possibilities that can accommodate more header fields.

For every switch, we maintain two binary trees: one based on source IP addresses and the other based on destination IP addresses. Every node corresponds to a source or destination address prefix. When the controller wants to install a rule r to a switch, it adds the rule action to both the source and destination trees at the nodes corresponding to the source and destination prefix included in r.

Given this binary tree based representation of rules installed at a switch, we aggregate rules as follows. Consider a new rule r added at nodes s and d in the source and destination trees, respectively. We can potentially aggregate if r has the same action as another rule r' and if r' satisfies one of the following conditions in both the source and destination trees: 1) r and r' are at the same node in the tree, 2) r' is r's parent, or 3) r and r' are siblings. Moreover, in the case that r is aggregated up to its parent in either tree, we recursively continue checking upwards in the source and destination trees to see if further opportunities for aggregation exist.

Figures 2 and 3 show a three-level sub-tree representing the last two bits of the IP space, along with example rules. Different colors represent different rule actions. First, rules r_2 and r_3 are aggregated into r_6 because they a) have the same associated action (blue), b) are at the same node in the destination tree, and c) are siblings in the source tree. Thereafter, recursive checks for aggregation find that r_1 and r_6 can be aggregated into r_7. On the other hand, though r_4 and r_5 have the same action (red) and are siblings in the source tree, they cannot be aggregated since they do not satisfy any one of criteria 1), 2), and 3) mentioned above.

5.2 Aggressive aggregation

As described so far, we can aggregate a rule r up to its parent node only if there exists another rule with the same action at r's sibling. This limits the ability to aggregate similar rules when two rules are not at the same node or share a parent, but share a common ancestor. For example, in Figure 2, although r_4 and r_5 could not be aggregated because they do not have a common parent in the destination tree, they could potentially be aggregated up to their common grandparent.

However, unless we place any restrictions, aggregating rules with common ancestors could result in the aggregation of very dissimilar rules. For example, two rules that are at the leftmost and rightmost nodes in either tree (as dissimilar as they can get), can be aggregated up to their common ancestor—the root. In such cases, the aggregated rule will span a very large part of the IP address space, and matched packets will be associated with an action that is perhaps not intended by the application policy.

To limit the aggressiveness of aggregation with common ancestors, we use a threshold T. We install an aggregated rule at a node in the source or destination tree only if the controller has already inserted rules that are associated with at least $T\%$ of the leaves in the subtree rooted at the node. For example, in Figure 2, we could aggregate r_4 and r_5 into the root of the destination tree if $T \geq 50\%$.

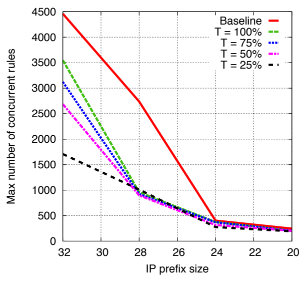
One of the side-effects of aggressive aggregation is that it can violate application policies. When threshold-based aggregation is used, an aggregated rule may match packets that are not covered by rules previously installed by the controller. In the absence of the aggregated rule, these packets would trigger a PacketIn message sent to the controller, to which the controller may have chosen to insert a rule with a different action than the aggregated rule. Later, we evaluate the extent to which policy violations occur and the trade-offs involved in eliminating them.

5.3 Evaluation

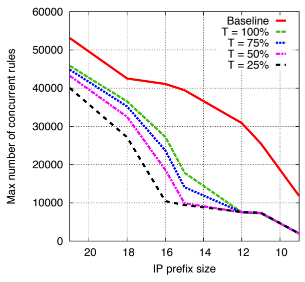
Table 2 shows the results of our measurement.

Rule savings of simple aggregation. Figure 4 shows how the rule savings vary with the use of wildcards *i.e.*, reducing the IP prefix size (ignore the lines for T < 100% for now). In the *Abilene* dataset, as we decrease the prefix size, the potential for aggregation increases. Without aggregation, specifying rules at /16 granularity (rather than /21) reduces their number to only around 40K (compared to slightly over 50K). In contrast, when using aggregation, the maximum number of rules is further reduced by third (to around 25K). The savings are even bigger for the *Campus* data set: up to 62% savings when aggregating at /28 prefix).

Overhead of simple aggregation. Aggregation may increase the number of switch operations, because one rule addition or deletion performed by the controller can translate to several operations at the switch. This is reflected in the *Abilene* data where the



(a)



(b)

Fig. 4: Maximum number of concurrent rules needed to cover the (a) *Campus* and (b) *Abilene* flows, as we vary the value of T and use wildcards.

operation rate increases slightly by 5% (see Table 2). However, when we have many aggregations, we may also save operations because we delete an aggregated rule from the switch only when all rules it aggregates are deleted. Since the *Campus* data has more rule savings (and implicitly more higher-in-the-tree aggregations), the number of operations decreases slightly by 3%.

Is aggressive aggregation effective? Table 2 and Figure 4 show that aggressive aggregation can reduce dramatically the number of rules (by 62% for *Campus* and 24% for *Abilene*) and the rate of switch operations (45% for *Campus* and 2% for *Abilene*). Using a threshold has only limited effect on the wildcarded *Campus* rules. When the prefix size is big, the savings are significant (up to 62% with /28 prefix and 75% threshold). However, because the IPs in the *Campus* data are more similar, most rules are already aggregated when the prefix size decreases enough (less than /24) and using a threshold cannot yield further savings.

We measure policy violations as the percentage of flows that are forwarded with a different action when we aggregate rules compared to a deployment where there is no aggregation. The fraction of flows for which rule aggregation leads to an incorrect output action is low. When the threshold is 25% *i.e.*, we install an aggregate rule in a node even when only a quarter of the leafs in its subtree have an associated rule, less than 1% of the *Abilene* flows could be misdirected. The number of policy violations decreases with higher thresholds. There are no violations for *Campus*, as the set of output actions is less varied than for *Abilene*.

5.4 Summary

Automatically aggregating similar rules reduces their number by up to 20% compared to IP-only rules with 60s timeout at negligible changes in control channel overhead. Operators or programmers can further increase efficiency (up to 62% rule reduction) if they allow a small part of the traffic (under 1%) to be directed to other destinations. While this is unacceptable for most applications, it may be a solution for dedicated network deployments where any of a set of destinations is acceptable (*e.g.*, load balancers, firewalls, anycast). As Table 2 shows, for many cases, it is more effective to use small timeouts than any automatic aggregation.

6 Conclusions and Future Work

Our real-world traces study shows that simple OpenFlow-based mechanisms, such as lowering rule expiration timeouts, are effective in managing the configuration size on OpenFlow switches although may increase (sometimes unacceptably) the utilization of the switch-to-controller channel. Other manual (using wildcards) or automatic (aggregating similar rules) mechanisms may reduce the size of the rule set even higher but curtail the expressiveness of the high-level policy and may, in a small number of cases, misdirect some packets. Understanding these trade-offs is important to SDN operators and programmers that must write network policies that satisfy both infrastructure and application constraints.

Our ongoing and future work spans two directions. On one hand, we are studying the adaptability of existing IP-based rule compression mechanisms [17] to OpenFlow. We are exploring the use of R-trees [9] to extend our ability to identify and aggregate rule similar in fields other than IP addresses (*e.g.*, protocol).

References

1. NEC OpenFlow switches. `http://www.openflow.org/wp/switch-NEC/`.
2. Pronto OpenFlow switches. `http://www.openflow.org/wp/switch-Pronto/`.
3. M. Appelman and M. D. Boer. Performance Analysis of OpenFlow Hardware. Technical report, University of Amsterdam, 2012.
4. T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.
5. A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalag, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *SIGCOMM*, 2011.
6. Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet Classifiers in Ternary CAMs Can Be Smaller. In *ACM Sigmetrics*, 2006.
7. N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A netowrk programming language. In *ACM IFIP*, 2011.
8. A. D. Freguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control in SDNs. In *SIGCOMM*, 2013.
9. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
10. HP 3800. `http://h17007.www1.hp.com/us/en/networking/products/switches/HP_3800_Switch_Series/index.aspx`.
11. IBM OpenFlow switches. `http://www.openflow.org/wp/ibm-switch/`.
12. S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurement and analysis. In *IMC*, 2009.
13. N. Katta, O. Alipourfad, J. Rexford, and D. Walker. Infinite CacheFlow in Software-Defined Networks. In *HotSDN*, 2014.
14. K. Kogan, S. Nikolenko, W. Culhane, P. Eugster, and E. Ruan. Towards Efficient Implementation of Packet Classifiers in SDN/OpenFlow. In *HotSDN*, 2013.
15. H. Liu. Routing table compaction in ternary CAM. *IEEE Micro*, 22(1):55–64, 2002.
16. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM Sigcomm CCR*, 38:69–74, 2008.
17. C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. 18(2), 2010.
18. C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programs. In *ACM POPL*, 2012.
19. C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *NSDI*, 2013.
20. M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *NSDI*, 2013.
21. Openflow multipath proposal. `http://www.openflow.org/wk/index.php/Multipath_Proposal`.
22. Openflow switch specification, 1.0.0. `http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf`.
23. V. C. Ravikumar and R. N. Mahapatra. TCAM architecture for IP lookup using prefix properties. *IEEE Micro*, 24(2):60–69, 2004.
24. C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *PAM*, 2012.
25. N. Sarrar, R. Wuttke, S. Schmid, M. Bienkowski, and S. Uhlig. Leveraging Locality for FIB Aggregation. In *IEEE Globecom*, 2014.
26. R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.
27. M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *ACM Sigcomm*, 2010.