TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 07/10 pp196-209
Volume 21, Number 2, April 2016

HBD: Towards Efficient Reactive Rule Dispatching in Software-Defined Networks

Chang Chen, Xiaohe Hu, Kai Zheng, Xiang Wang, Yang Xiang, and Jun Li*

Abstract: Most types of Software-Defined Networking (SDN) architectures employ reactive rule dispatching to enhance real-time network control. The rule dispatcher, as one of the key components of the network controller, generates and dispatches the cache rules with response for the packet-in messages from the forwarding devices. It is important not only for ensuring semantic integrity between the control plane and the data plane, but also for preserving the performance and efficiency of the forwarding devices. In theory, generating the optimal cache rules on demands is a knotty problem due to its high theoretical complexity. In practice, however, the characteristics lying in real-life traffic and rule sets demonstrate that temporal and spacial localities can be leveraged by the rule dispatcher to significantly reduce computational overhead. In this paper, we take a deep-dive into the reactive rule dispatching problem through modeling and complexity analysis, and then we propose a set of algorithms named Hierarchy-Based Dispatching (HBD), which exploits the nesting hierarchy of rules to simplify the theoretical model of the problem, and trade the strict coverage optimality off for a more practical but still superior rule generation result. Experimental result shows that HBD achieves performance gain in terms of rule cache capability and rule storage efficiency against the existing approaches.

Key words: Software-Defined Networking (SDN); reactive rule dispatching; rule cache; performance

1 Introduction

The ultimate goal of networks today is to improve traffic processing performance and reduce network deployment and operation cost, while satisfying diverse policies for network control. These policies are mostly presented in the form of packet processing rules installed in the forwarding devices, which process the incoming packets according to values of specific fields in their headers.

To better support the growth and innovation of networks, Software-Defined Networking (SDN) offloads the control logic from the forwarding devices to a separate and centralized control plane. Correspondingly, control policy determining is thus separated from data-plane packet processing, and this makes the packet processing rules in the forwarding devices logically become the *cache* of control policies in the controllers. Under this clear division of roles, an urgent issue regarding the execution of control policies emerges: in respond to the data plane's rule installation requests (i.e., packet-in message[1]), how does the controller reactively generate proper rules and dispatch them to the corresponding devices, while preserving

• Chang Chen, Xiaohe Hu, and Xiang Wang are with Department of Automation, Research Institute of Information Technology, Tsinghua University, Beijing 100084, China. E-mail: {chenchang13, hu-xh14, xiang-wang11}@mails.tsinghua.edu.cn.
• Kai Zheng is with IBM China Research Lab, Beijing 100084, China. E-mail: zhengkai@cn.ibm.com.
• Yang Xiang is with Research Institute of Information Technology, Tsinghua University, Beijing 100084, China. E-mail: sharang@tsinghua.edu.cn.
• Jun Li is with Research Institute of Information Technology, Tsinghua National Lab for Information Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: junl@tsinghua.edu.cn.
∗ To whom correspondence should be addressed.
Manuscript received: 2015-03-31; revised: 2015-08-25; accepted: 2015-08-28

semantic integrity (i.e., decision consistency of control plane and data plane) and optimizing efficiency? We call it the *reactive rule dispatching* problem.

To realize an elastic SDN with fine-grained control, reactive rule dispatching is the basis of dealing with complex and dynamic events (e.g., host mobility, abnormal traffic analysis, and failure recovery). Although rules can, alternatively, be proactively pushed into the data plane[2–5] before specific traffic comes, however, due to the expensive and limited fast data-plane flow table, it is costly or even infeasible for the forwarding devices to cache potential rules all at the bootstrapping stage. Fortunately, network traffic has both temporal and spatial localities in most scenarios[6], which makes it possible for reactive rule dispatching to perform well by optimal caching of rules.

Figure 1 shows the general architecture of reactive rule dispatching in an OpenFlow-based SDN. As a core module of the controller, the rule dispatcher logically sits between the rule matching module in the controller, which holds the entire control rules, and the rule table (or flow table) in the forwarding device, which caches the recently dispatched rules. To handle a "cache miss" message (i.e., a packet-in message) for certain traffic flow, it is performance critical for the rule dispatcher to determine which or what rule(s) to be dispatched to the forwarding device, not being too generic (i.e., with too much wildcards or too large ranges to lead to semantic error) or too specific (i.e., limiting the probability of covering more upcoming traffic and leading to high cache miss ratio).

Using some straightforward approaches to preserve decision consistency, reactive rule dispatching is implemented in various types of SDN architectures and systems[2, 7–9]. However, all these straightforward approaches (which will be detailed in Section 2) lack the consideration of performance. For example, they usually generate exactly matching rules for the traffic on a per flow basis and thus accompany with remarkably high cache miss ratio.

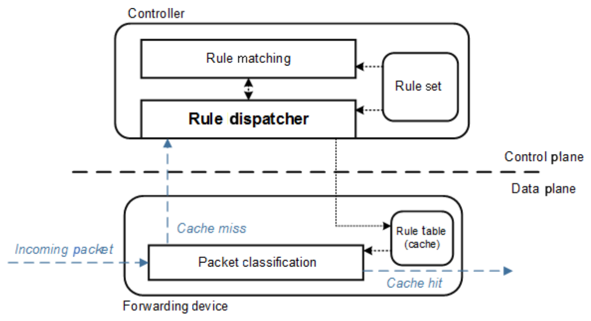


Fig. 1 General model of reactive rule dispatching.

From a high level, an optimized cache rule generation method should take into consideration the following performance criteria:

(1) Low latency: Generally, in the case when the rule required for the specific packet to forward resides locally in the forwarding device (i.e., in the case of cache hit), the forwarding time is on the order of nanosecond. In contrast, the processing latency in the case of a cache miss is on the order of millisecond[10]. The high time penalty of a cache miss comes essentially from network transmission overhead between the controller and the forwarding devices. Obviously, the higher processing latency on the orders of magnitude suggests that the dispatched cache rules should cover as much potential upcoming traffic as possible (e.g., with more wildcard fields or larger range to match).

(2) Efficient memory utilization: The feasibility of implementation can be enhanced if the generated cache rules make optimal use of the limited memory space in forwarding devices. For example, Ternary Content Addressable Memory (TCAM) is one of the popular lookup mechanisms for packet classification. However, as TCAM is space-limited, expensive, and power-hungry, the commodity switches usually support only thousands to tens of thousands of rules. Under these constrains, the rule dispatcher should also be responsible to ensure the efficient use of storage resource in forwarding devices.

Reactive rule dispatching is inherently hard to optimize due to the existence of rule dependency (caused by rule overlap and rule priority). Simply generating and dispatching cache rules regardless of rule dependency may cause the action inconsistency between the full rule set and the cache rules. On the other hand, finding the *optimal* cache rule regarding high traffic coverage (a.k.a., high cache hit ratio) implies high computational complexity in theory, which is usually unacceptable for real-time packet processing. For the sake of feasibility, it is necessary for the rule dispatching algorithms to exploit the inherent characteristics lying in real-life rule sets and traffic, and to flexibly trade strict optimality off for a more practical but still superior solution. In general, the proposed work makes the following main contributions:

Problem deep-dive: The geometrical model and complexity of rule dispatching problem are studied in this paper, along with the summary and classification of existing works according to different design decisions.

Heuristic rule dispatching algorithm: Based on the observation of characteristics of real-life rule sets, we propose Hierarchy-Based Dispatching (HBD), including a set of algorithms for determining the near-optimal cache rules according to packet-in messages. Besides, the cache rules dispatched by HBD have an addtional bonus attribute: Any pair of the cache rules is either disjointed or harmlessly overlapped (i.e., the two cache rules overlap with each other but are derived from a same original rule). This nice attribute[11] can be leveraged accompaning with various kinds of packet classification schemes for faster cache rule lookup on the data plane.

Performance evaluation: Based on both synthetic and real-life rules, the proposed algorithms as well as several existing approaches are evaluated in terms of performance metrics such as cache hit rate, traffic coverage capability, and processing time.

The rest of this paper is organized as follows: The related works are reviewed in Section 2. In Section 3 we introduce the optimization objective and modeling of reactive rule dispatching problem. In Section 4 we discuss several design principles in detail, and give a short study on the overlap characteristic of real-life rules. The proposed algorithms are presented in Section 5 and evaluated in Section 6. In Section 7 we conclude our work.

2 Related Work

2.1 Straightforward approaches

The straightforward rule dispatching approaches applied in prior SDN architectures can be summarized as follows: (1) Exact dispatching: Dispatching exact-match rule for the specific packet to forward, which is employed in several prior SDN architectures such as Ethane[7] and DevoFlow[10]. This approach suffers from high cache miss ratio due to the weak traffic coverage capability of exact-match rules. (2) Abundant dispatching: Dispatching the rule matching the specific packet along with all the dependent rules (where the dependency chain of rules is studied in Ref. [12]). This approach leads to excessive rule space usage in devices and in some cases, many of the dispatched dependent rules do not help to hit the subsequent traffic. (3) Fragmented dispatching: Transforming interdependent rules into a set of disjointed sub-rules before dispatching. This approach has relatively nice traffic coverage but the overlaps between multi-dimensional rules may lead to the explosion of the population of the transformed sub-rules[13].

2.2 Prior algorithms

Unlike the straightforward approaches mainly focusing on semantic correctness of the dispatched rules, several prior arts also target at meeting the performance requirements through well-designed rule dispatching algorithms, which can be summarized according to different design decisions:

(1) Evolving-based algorithm: In evolving-based algorithms, the boundaries of the dispatched rules are dynamically expanded (i.e., evolved) according to incoming traffic. Though they are designed for the scenario of two-stage rule matching schemes inside a packet forwarding device, the proposed algorithms are still enlightening today's SDN scenarios. In Ref. [14], Smart Rule Cache (SRC) introduces a representative evolving cache rule construction algorithm, which is also applied in DIFANE[2]. Another evolving-based algorithm is introduced in Storm[15], which proposes a rule caching system for software routers. However, the performance of SRC and Storm relies too much on the traffic pattern. For example, every single packet in an address-incrementing SYN flood traffic may trigger a cache miss because of the incremental extension of rule boundaries. Besides, their high cache hit ratios to some extent result from the aggregation of rules with the same action, which leads to incapability of preserving statistical information for each individual flow.

(2) Dependency-based algorithm: As an improvement to the straightforward abundant dispatching approach, the Cover-set algorithm presented in CacheFlow[12] finds each rule's immediate ancestor rules in the dependency graph as the "cover-set" of the rule, and dispatches rules associated with their cover-sets to preserve semantic correctness. In the CacheFlow architecture, the Cover-set algorithm is used to proactively dispatch to the hardware switches a group of rules with high "weights" and low "costs". At run-time, the packets that match the cover-sets are redirected to some software switches for further handling.

(3) Cut-based algorithm: As one latest work targeting at reactive rule dispatching, CAching in Buckets (CAB)[16] dispatches a set of buckets associated with their overlapped rules, and implements a two-stage table pipeline in the forwarding device. The bucket generation procedure, which directly affects cache performance, shares similarity with space decomposition based packet classification algorithms such as HiCuts[17] and HyperCuts[18].

The advantages of the proposed approaches against the prior-arts will be discussed later in Section 4.

3 Problem Statement

To make a clear view of reactive rule dispatching problem, we introduce the theoretical model, set the optimization objective, and propose an optimal solution to the problem in this section.

3.1 Modeling

As shown in Fig. 1, the forwarding devices rely on packet classification to associate the forwarding rules or actions with the traffic. Formally speaking, a packet is classified according to some specific header fields values (e.g., the typical 5-tuples, source and destination IP addresses, source and destination ports, and protocol number). From a geometrical point of view, a packet $P = \{p_1, p_2, \ldots, p_D\}$ is a point in a D-dimensional address space S, where D is the number of header fields required to classify the packets, and the d-th field value of P is denoted as p_d. A rule $R = \{[r_{1\text{s}}, r_{1\text{e}}], [r_{2\text{s}}, r_{2\text{e}}], \ldots, [r_{D\text{s}}, r_{D\text{e}}]\}$ also contains D components. The d-th component $R[d] = [r_{d\text{s}}, r_{d\text{e}}]$ refers to a range in the d-th dimension of S, and all the D ranges of R compose a D-dimensional hyper-rectangle.

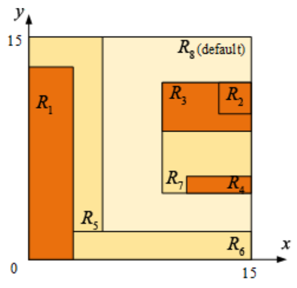
Packet classification can be regarded as a point location problem in computational geometry. Accordingly, the process of generating an optimal dispatched rule can be considered as finding the largest inscribed hyper-rectangle for a given point in the address space.

Figure 2a shows a 2-dimensional rule set example and Fig. 2b is the corresponding geometrical model. There are 8 rules in a 2-dimensional address space. For packet classification, some of the rules overlap with each other and the priority determines the final match for packets in the overlapped regions. As illustrated in Fig. 2c, given the packet $P = \{12, 7\}$ which matches R_7, the rule dispatcher needs to generate and dispatch to the forwarding device a proper rule.

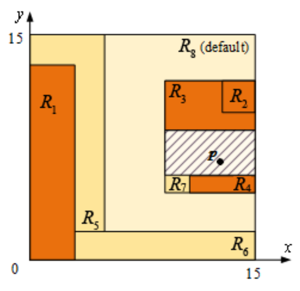
Due to rule overlapping and different priorities for the corresponding forwarding actions, simply dispatching

Rule	Field-x	Field-y	Priority	Action
R_1	[0, 3]	[0, 13]	1	Action 1
R_2	[13, 15]	[10, 12]	5	Action 2
R_3	[8, 15]	[9, 12]	9	Action 3
R_4	[10, 15]	[5, 6]	10	Action 4
R_5	[0, 5]	[3, 15]	37	Action 5
R_6	[0, 15]	[0, 2]	120	Action 6
R_7	[8, 15]	[5, 12]	200	Action 7
R_8	[0, 15]	[0, 15]	999	Action 8

(a) An example rule set



(b) The geometrical model



(c) The optimal cache rule for packet P

Fig. 2 A two-dimensional example.

the intact rule $R_7 = [8, 15], [5, 12]$ might break the action consistency (For example, in the space covered by R_4, which is nested in R_7, the action associated with R_4 should be taken instead of that of R_7 according to their priority; thus simply applying the action of R_7 to the entire hyper-rectangle covered by R_7 may lead to loss of consistency). As a straightforward idea, "exact dispatching" is to dispatch the very specific rule representing only the single point as the packet header P indicates directly, i.e., $[12, 12], [7, 7]$. This obviously preserves the action consistency but may suffer from high cache miss ratio due to the very low coverage of address space. In contrast, "abundant dispatching" is to dispatch all the dependent rules (e.g., R_7, R_2, R_3, and R_4 in our example), which occupies excessive rule space in the underlying forwarding device. Moreover, some of the "remote" dispatched rules (e.g., R_2) may not actually help with increasing cache hit ratio due to traffic's spatial locality.

Suppose that the dispatcher only generates one cache rule. The optimal rule to generate should potentially cover the maximum space in which the corresponding rule action is with the highest priority. As shown by the shaded rectangle in Fig. 2c, the optimal cache rule for the example is $[8, 15], [7, 8]$, which is the largest inscribed hyper-rectangle that preserves the action consistency.

3.2 Optimization objective

In an SDN with reactive rule dispatching, the average time for processing a packet can be approximated by

$$T_{\text{avrg}} = T_{\text{hit}} + \eta_{\text{miss}} \cdot (T_{\text{trans}} + T_{\text{gen}}) \tag{1}$$

where T_{hit} is the average rule lookup time in the forwarding device, η_{miss} is the rule cache miss ratio, T_{trans} is the average delay of the transmission between two planes, and T_{gen} is the average time for generating a cache rule.

The optimization objective of rule dispatching includes two aspects:

(1) To reduce T_{avrg} according to

$$T_{\text{avrg}} = T_{\text{hit}} + T_{\text{trans}} \cdot \eta_{\text{miss}} \cdot \left(1 + \frac{T_{\text{gen}}}{T_{\text{trans}}}\right) \tag{2}$$

Generally, T_{hit} is on the order of nanosecond and T_{trans} is on the order of millisecond[10]. For the other two arguments, it is critical to reduce η_{miss} for the obtimization objective, while T_{gen} is the cost to achieve this. In order to ensure that the benefits outweigh the cost, the rule dispatching algorithm needs to produce high-coverage rules efficiently; it is equivalent with reducing η_{miss} and guaranteeing $T_{\text{gen}} \ll T_{\text{trans}}$.

(2) To satisfy the rule space constraints in the forwarding devices (e.g., the TCAM table size constraint): The reduction of η_{miss} should not rely merely on installing larger number of rules on the devices. Instead, the focus should be on leveraging the temporal and spacial localities of the traffic and generating less but higher coverage rules (i.e., as more wildcards or larger ranges in the generated rules as possible while preserving the semantic consistency to the control rules).

3.3 The optimal solution and complexity analysis

In this section, we brief an optimal solution to the problem of determining a single optimal cache rule and analyze its complexity.

Let point P represent the input packet, hyper-rectangle R_{match} represent the highest priority rule that matches P, and R represent the optimal cache rule we wish to generate. Suppose R is initialized to be the exact-point rule located at P. The mission is to keep expanding and adjusting R's boundary (i.e., $[r_{ds}, r_{de}]$ of R, $d = 1, 2, \ldots, D$) until finding the largest possible hyper-rectangle, which represents the optimal cache rule. There are three constrains of the expansion of R:

(1) R covers P:

$$\forall d \in [1, D] : p_d \in R[d] \tag{3}$$

(2) R is nested in R_{match}, that is, R_{match} forms a hard limit for the boundary expansion of R:

$$\forall d \in [1, D] : R[d] \subseteq R_{\text{match}}[d] \tag{4}$$

(3) The rules that overlap with R_{match} and have higher priorities, referred to as R^*, must not overlap with R; in other word, R^* forms the obstacles for the boundary expansion of R:

$$\forall d \in [1, D] : R[d] \cap R^*[d] = \varnothing \tag{5}$$

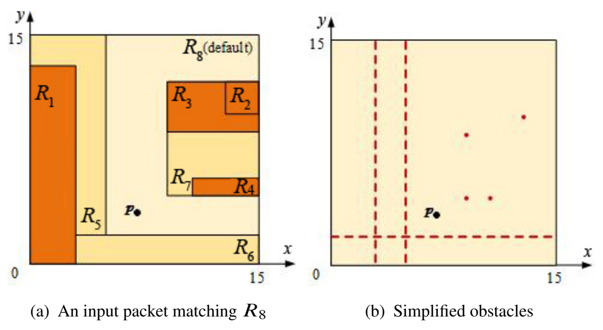
For each R^*, if there are exactly M $(0 \leqslant M < D)$ dimensions in which the range of R^* covers the projection of P (i.e., $p_d \in R[d]$), R_{match} can be simplified as an M-dimensional obstacle, which provides $D - M$ possibilities for the boundary adjustment of R. For example, in Fig. 3a, the matching rule $R_{\text{match}} = R_8$, the obstacle rules $R^* = R_1 - R_7$. As shown in Fig. 3b, R_8 forms R's boundary limit. R_1, R_5, and R_6 are simplified as three 1-dimensional obstacles (lines), each of which provides only one choice for boundary expansion of R (field-x for R_1 and R_5, field-y for R_6). R_2, R_3, R_4, and R_7 are simplified as four 0-dimensional obstacles (points), each of which provides two choices for boundary adjustment (field-x or field-y).

In the worst case, all the obstacles are points, and all possible combinations of boundaries need to be examined. Thus the optimal solution requires extra $\Theta(ND)$ memory space and $\Theta(D^N)$ computation time, where N is the number of rules that overlap with R_{match} and also have higher priorities. As we can see, the complexity of finding the optimal cache rule is unacceptable in practice.

4 Design Decision

4.1 Design principles

Compared with the prior-arts, the proposed approach is based on four distinct principles:



(a) An input packet matching R_8 (b) Simplified obstacles

Fig. 3 Finding the optimal cache rule.

(1) Constructing new cache rules rather than preserving original ones: Unlike Cover-set algorithm that determines "which rule to be dispatched", we determine "what rule to be dispatched". This makes it more possible to customize and therefore optimize the dispatched rule. Besides, every reconstructed rule in our approach is derived from an original rule, thus the action and more attributes (e.g., counters) of the original rule can be inherited directly.

(2) Dispatching only one rule per "packet-in": According to the observation on the spatial locality of real-life traffic, the nearer the distance to the "packet-in" point, the more possible the dispatched rule may hit the future traffic. Thus we argue that dispatching multiple overlapping-but-sparse rules to the forwarding device will have little advantages yet incur significant storage overhead. Therefore, in the proposed rule dispatching scheme, we concentrate on finding the "best" single rule which covers the largest "inscribed" hyper-rectangle for a given "packet-in" point in the address space.

(3) No assumption on device functionality: Unlike CAB that requires the forwarding device to support two-stage flow table pipeline (i.e., to be compatible with OpenFlow specification), we make no assumption on device functionality. Our approach should be compatible with the legacy forwarding devices with any kind of packet classification approach applied.

According to different design decisions, we compare the main attributes of existing works along with our proposed approach in Table 1.

(4) Adapted to the diversity of rule sets characteristics: As mentioned before, the challenges of the rule dispatching problem comes from the existence of rule overlap and rule priority. We do not assume that the control rules are basically disjointed or, conversely, seriously overlapped. As a matter of fact, different kinds of rule set tend to form different kinds of overlap characteristics. For example, the traditional routing or switching rules are more likely to be non-overlapping, but the ACL rules at a multi-tier network are more likely to be overlapped with each other because of the differentiated permissions to different subnets or individuals.

Table 1 Existing work.

Approach	Rule recons-truction	Number of rules dispatched per packet-in	Assumption on forwarding device
Exact	—	Single	No
Abundant	No	Multiple	No
Fragmented	Yes	Single	No
Evolving-based	Yes	Single	—
Cover-set	No	Multiple	Need extra software switches
CAB	No	Multiple	Need two-stage flow table
Our work	Yes	Single	No

Since rule overlap cannot be avoided, in the next section we study the rule overlap characteristics in several real-life rule sets, in order to further introduce the heuristic to the algorithm for simplifying the cache rule generation process.

4.2 Overlap characteristic

The overlap characteristic is studied using four representive real-life rule sets. Two of the rule sets, ACL1 and FW1, are publicly available from Ref. [19]. The other two rule sets, vAccess and vGateway, are extracted from the network controller and the security controller of a cloud management platform, respectively. This platform[20, 21] provides secure virtual private cloud service in a Tier-IV datacenter. vAccess is for the access software switch in a physical server, containing 1982 OpenFlow rules; and vGateway is for a tenant virtual gateway that delivers routing and security services among multiple virtual networks, containing 3106 five-tuple rules.

To study the overlap degree of these rule sets, we define the number of overlap tiers as the maximum depth of the rule set's dependency chains[12]. (The overlap tiers can be built in a bottom-up traverse of the rule set. The default rule forms the bottom tier. Any two rules in the same tier must not overlap with each other.) Table 2 shows the number of overlap tiers of each rule set and suggests the diversity of overlap degree of different kinds of rules.

Moreover, we define three relations between two overlapped rules (R_1 and R_2) as follow:

(1) Nested: R_1 is nested in R_2 if

$$\forall d \in [1, D] : R_1[d] \subseteq R_2[d] \tag{6}$$

We call R_1 the subset rule and R_2 the superset rule.

Table 2 The four real-life rule sets.

Set	Number of rules	Number of overlap-tiers
ACL1	753	38
FW1	270	5
vAccess	1982	4
vGateway	3106	11

(2) Crossed: R_1 and R_2 are crossed if

$$\begin{array}{l}\exists d_1, d_2 \in [1, D]: \\ R_1[d_1] \subset R_2[d_1] \text{ and } R_1[d_2] \subset R_2[d_2]\end{array} \tag{7}$$

(3) Partially overlapped: The other cases.

Suppose that all the rules can be organized in a complete "nesting hierarchy", namely, for each pair of rules, they are either disjointed or nested in priority order (i.e., the higher-priority one is nested in the lower-priority one). For the optimal solution presented in Section 3, the number of "obstacle" rules needed to be examed during the boundary adjustment of R can be significantly reduced. It is because that after examing a "obstacle" superset rule, all the corresponding subset rules nested in this superset rule will certainly not affect the further boundary adjustment of R, and thus can be ignored.

For the purpose of observing the nesting characteristic, we further provide a more intense analysis on overlap type using ACL1, which is the worst case among the four rule sets. Figure 4 presents the overlap characteristic of ACL1 in detail. The x-axis represents the ID of the rules; the rule with a smaller ID has a higher priority. The blue curve shows the total number of rules overlapped with a specific rule, the green curve shows the number of higher priority rules overlapped with a specific rule (i.e., the rules that might cause action inconsistency), while the red curve shows the number of rules overlapped but non-nested with a specific rule. We can see that: (1) For a specific rule R, the number of higher-priority rules that overlap with R might be numerous, especially for the large-sized low-priority rules; (2) Though most of the overlapped rule pairs tend to be "nested", the "crossed" or "partially overlapped" relations still exist among rules.

5 Hierarchy-Based Dispatching

The proposed HBD approach includes the following three threads: (1) Pre-processing: The rule dispatcher divides all the control rules into several groups, such that each of the groups forms a complete nesting hierarchy in the address space, respectively. (2) Cache rule generation: With the nesting hierarchy trees built in pre-processing, the rule dispatcher reactively generates high-coverage cache rules using a greedy algorithm at run-time. (3) Rule set update: The rule dispatcher supports dynamic rule set update by means of incrementally updating the nesting hierarchy trees.

5.1 Pre-processing

According to the observation on overlap characteristics, we cannot assume that the rule set is normally organized in a complete nesting hierarchy. In the example of Fig. 5a, even if most rules form a nesting hierarchy, there are still some rule pairs that are crossed or partially overlapped, including R_5 and R_7, R_6 and R_8, R_7 and R_8, and R_8 and R_9. The rule that is crossed or partially overlapped with another rule is reffered to as a "saboteur rule" in the following.

In order to build the nesting hierarchy of the rule set where all saboteur rules are eliminated, HBD employs a "grouping + decomposing" strategy for pre-processing.

5.1.1 The naive decomposing strategy

One intuitive solution to handle saboteur rules is to decompose them into some sub-rules until no saboteurs exist. Using this strategy, the operation of building a nesting hierarchy can be accomplished in a single bottom-up traverse of the rule set. Algorithm 1 shows the process of building the nesting hierarchy (which is organized as a tree structure). During the bottom-

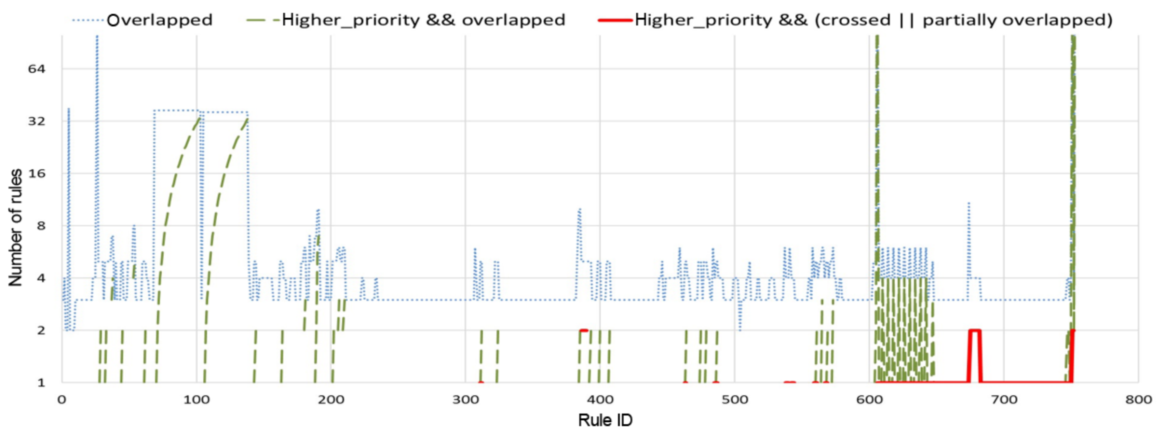
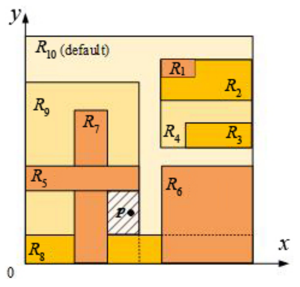
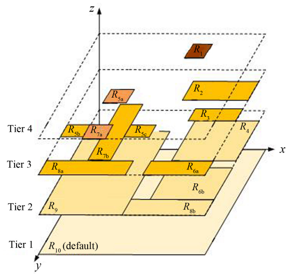


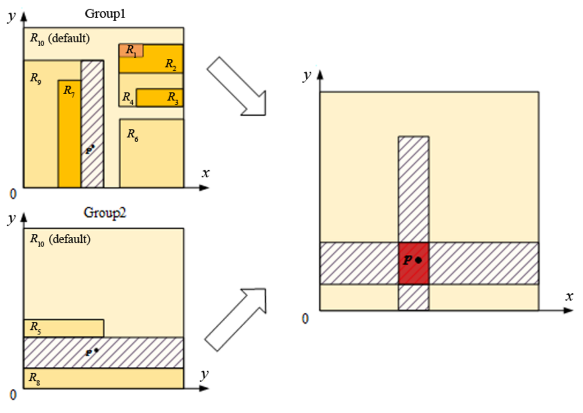
Fig. 4 Overlap characteristic of ACL1.



(a) An example rule set



(b) The naive decomposing strategy



(c) Grouping-based nesting hierarchy

Fig. 5 A two-dimensional example.

Algorithm 1 Decomposing-based pre-processing

```
1: function BUILDHIERARCHY_DECOMPOSING(Ruleset)
2:     stack = {}
3:     for each $R$ in Ruleset (decending priority) do
4:         stack.push($R$)
5:     end for
6:     tree = {stack.pop()}
7:     while stack not empty do
8:         $R$ = stack.pop()
9:         for each $R_t$ in tree do
10:            if $R$ is nested in $R_t$ then
11:                $R_t$.addChild($R$)
12:                continue
13:            end if
14:            if $R_t$.is Saboteur($R$) then
15:                $R'$ = spaceIntersection($R, R_t$)
16:                $R_t$.addChild($R'$)
17:                stack.push($R - R'$)
18:            end if
19:        end for
20:    end while
21:    return tree
22: end function
```

up traverse, when a new saboteur rule (i.e., crossed or partially overlapped with some pre-traversed rule) is encountered, it will be decomposed until every sub-rule of it can find a certain pre-traversed rule to be its superset. The decomposing is achieved by function spaceIntersection(). Figure 5b shows the nesting hierarchy of the example rule set using this naive decomposing strategy. R_5, R_6, R_7, and R_8 are decomposed into several nested sub-rules, which afterwards meet the nesting hierarchy requirement.

However, the $R - R'$ step in line 17 might generate numerous sub-rules of R ($2D$ in the worst case), which causes indeterminacy to the number of rules after building the nesting hierarchy. As shown in Table 3, the "decomposing" column suggests the infeasibility of direct decomposition of all saboteur rules. For example, after the direct decomposition, the size of vGateway swells from 3106 rules to over 200 thousand rules.

5.1.2 Grouping strategy

Instead of directly decomposing the saboteur rules, the grouping strategy eliminates the potential risks of rule number explosion by means of partitioning the rules into different groups where no saboteur rule exists in each group. It should be noticed that the default rule which covers the entire address space needs to be assigned to every group.

The grouping strategy is based on the following observation. Suppose that the original rules are divided into K groups. Consequently, the original problem of finding large inscribed hyper-rectangle R for a packet-in point P can be partitioned into K sub-problems. The result (i.e., generated cache rule) to each sub-problem is referred to as R_i $(i = 1, 2, \ldots, K)$. The intersected space of all R_i forms the final result R to the original problem (i.e., $R = \cap R_i$), which can guarantee the correctness. Thus, when dealing with a cache miss, the cache rule generation algorithm (which will be described later) is executed on all groups (i.e., rule subsets) parallelly. After that, the dispatcher conducts a space intersection of all intermediate cache rules to form the final result.

Algorithm 2 shows the grouping operation as well as building each group's nesting hierarchy. Each tree

Table 3 Scalability of the two strategies.

Rule set	Number of rules	Decomposing	Grouping	
		Number of rules after building	Number of groups	Number of rules after building
ACL1	753	2338	3	755
FW1	270	154 257	5	274
vAccess	1982	2154	3	1984
vGateway	3106	234 874	5	3110

Algorithm 2 Grouping-based pre-processing

```
1: function BuildHierarchy_Grouping(Ruleset)
2:     stack = {}, list = {}
3:     for each $R$ in Ruleset (decending priority) do
4:         stack.push($R$)
5:     end for
6:     tree = {stack.pop()}
7:     while stack not empty do
8:         $R$ = stack.pop()
9:         for each $R_t$ in tree do
10:            if $R$ is nested in $R_t$ then
11:                $R_t$.addChild($R$)
12:                continue
13:            end if
14:            if $R_t$. is Saboteur($R$) then
15:                list.add($R$)
16:            end if
17:        end for
18:    end while
19:    add tree to trees_list
20:    if list not empty then
21:        list.reverse(); list.add($R_{\text{default}}$)
22:        BuildHierarchy_Grouping(list)
23:    end if
24: end function
```

represents a group, and the trees_list contains all the trees (groups). The grouping procedure is implemented as a recursive function: when a saboteur rule is encountered, it is excluded from current group, and will be explored in later iterations. The algorithm terminates until in each group the rules form a complete nesting hierarchy. Figure 5c shows the grouping result of the example rule set, where the original rules are divided into two groups, and in each group the rules form a complete nesting hierarchy. The intersected space of the cache rule results of the two groups (as shown in red) represents the final cache rule result (the properties of the cache rule are inherited from R_9, which is with the higher priority). The "grouping" column in Table 3 shows the number of groups needed for eliminating saboteurs completely, which is small enough for reliable performance. However we can not assume that all rule sets in real-life have similar characteristics. To support the limitation of the group number (especially for large-sized rule sets), we combine the grouping and decomposing strategies to enhance the feasibility of implementation.

5.1.3 Combination of the two strategies

In the grouping process, the algorithm can introduce moderate rule decompositions in current group (instead of excluding all saboteur rules) based on the cost of decomposing a specific saboteur rule. The cost is defined as the number of sub-rules that would be produced by step $R - R'$ in Algorithm 1. Whether to decompose a saboteur rule or exclude it from current group depends on a pre-defined cost threshold: if the cost is not larger than the threshold, the algorithm chooses the former option; otherwise, the latter is chosen. For example, the cost of decomposing a rule that is partially overlapped on only one dimension might be 1, thus the algorithm decomposes the rule and keeps it in current group. But the cost of decomposing a rule crossed on multiple dimensions is usually high, thus this rule might be excluded from current group.

The question of how to set the cost threshold and the upper limit of group number leaves open. Generally, the upper limit of the group number should follow the maximum number that a particular platform supports to execute in parallel (e.g., 8 cores or threads), and the cost threshold depends on the overlap degree of rules. In our approach, the upper limit of group number is set to 4, and the cost threshold is set to 2.

5.2 Cache rule generation

With the rule set's nesting hierarchy, the cache rule generation process can be significantly simplified. Given the packet-in point P and its matching rule R, if R is on the topmost tier, the original R can be directly dispatched; otherwise, exploring only the immediate nested rules of R (i.e., the rules that the child nodes of R represent) is sufficient for preserving semantic integrity and optimizing performance. Thus in HBD the rule generation problem is simplified to a two-tier nesting model, containing only R and its immediate nested rules.

The theoretical optimal solution presented in Section 3 needs $\Theta(D^N)$ time to find the optimal cache rule because of D^N boundary adjustment possibilities in the worst case. It is impractical to apply this solution in practice due to the exponential computational complexity. Thus in HBD the optimality of coverage can be traded off for a much more efficient but still superior greedy algorithm. As shown in Algorithm 3, the output cache rule R_{out} is initialized to the packet's matching rule R. During the process of adjusting the boundary of R_{out}, R_{out} greedily reduces its boundary based on each immediate nested rule of R.

Algorithm 3 Cache rule generation

```
function CACHERULEGENERATE(packet, tree)
    $R_m$ = packet.match()
    if $R_m$ is a leaf node of tree then
        return $R_m$
    else
        $R_{\text{out}} = R_m$
        for each child node $R_{\text{obs}}$ of $R_m$ in tree do
            for each $d$ in $[1, D]$ do
                predict volume loss % : loss[$d$]
            end for
            choose $d$ with the minimum loss[$d$]
            adjust $R_{\text{out}}$ in the $d$-th dimension
        end for
        return $R_{\text{out}}$
    end if
end function
```

In contrast with the strict optimal solution, the greedy algorithm reduces the time complexity to $\Theta(ND)$. Additionally, the data size N is significantly reduced as the nesting hierarchy simplifies the problem model.

5.3 Rule set update

The nesting hierarchy naturally supports rule set update in the dynamic SDN scenarios. When adding a new set of rules to the rule set, the rule set update thread of HBD first uses Algorithm 2 to partition this new rules into several groups, each of which we refer to as a branch. Then the update thread incrementally adds each branch to the nesting hierarchy trees according to the location of the branch's root rule. The algorithm for adding a branch is shown in Algorithm 4.

As for a rule deletion request, HBD simply gives the rules descendants to its father node, and removes this rule node from current hierarchy tree.

5.4 Bonus attribute of HBD

The cache rules dispatched by HBD have an bonus attribute: Any pair of the cache rules are either disjointed or "harmlessly" overlapped (i.e., the two cache rules overlap with each other but are derived from a same original rule). It is called the order-independent attribute. The impact of this attribute is studied in SAX-PAC[11], one latest work that simplifies the classifier matching problem into several sub-problems. In general, the order-independent attribute can be leveraged by the packet processing module of forwarding devices in the following aspects: (1) The order-independence can eliminate all types of rule conflictions caused by rule-overlapping[22].

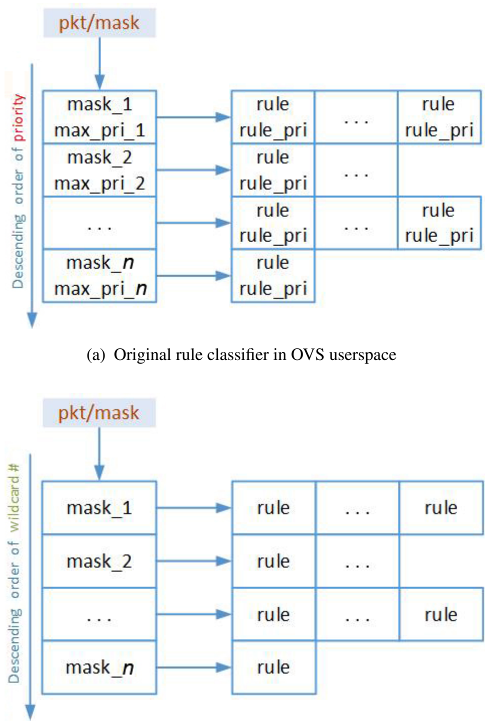
Algorithm 4 Add branch

```
function ADDBRANCH(branch, tree)
    $R$ = branch.root
    if $R$.leftcorner.match()! = $R$.rightcorner.match() then
        AddBranch(branch, next_tree)
        return
    else
        $R_m = R$.leftcorner.match()
    end if
    list = {}
    for each child node $R_c$ of $R_m$ in tree do
        if $R_c$ is nested in $R$ then
            list.add($R_c$)
        else if $R_c$ and $R$ are overlapped then
            AddBranch(branch, next_tree)
            return
        end if
    end for
    $R$.addChildren(list)
    $R_m$.deleteChildren(list)
    $R_m$.addChild($R$)
end function
```

(2) Order-independent rules are disjointed in the search space, which can improve both the spacial and time performances of most packet classification algorithms[23, 24]. (3) As the space that a certain rule locates is associated with only one single action, it is more convenient to track and debug these rules.

In this part, we brief an order-independent rule lookup method based on Open vSwitch (OVS)[25] for improving its packet processing performance. As shown in Fig. 6a, the dispatched rules are installed in the userspace of OVS and organized in a two-stage list structure. Originally, due to the dependency among rules, the classifier needs to organize the priority information and sort the former stage list in terms of the rule priorities. The main cost of the rule lookup process comes from two aspects: (1) the traverse of the two-stage list, which can hardly be pre-terminated due to rule dependency; and (2) the degradation of "megaflow"[25] that will be inserted to the kernel of OVS. Specifically, the megaflow (with many wildcards) is always degraded to a "microflow" (with no wildcards) when identifying the mask associated with the flow, which correspondingly degrades the packet process performance. It is because that the sorting method of the rule lists is bound with the rule priority and thus cannot be customized for performance consideration.

Using HBD, since the rules dispatched to OVS are order-independent, the traverse of the rule lists can be



(a) Original rule classifier in OVS userspace

(b) Simplification of the classifier

Fig. 6 Improvement on OVS classifier leveraging the order-independent attribute.

directly terminated when a rule matching occurs, and the OVS classifier no longer needs to sort the rule lists based on priorities. Furthermore, to maximum the wildcard number of megaflows, as shown in Fig. 6b, we sort the former stage list in descending order of wildcard number. Thus, the coverage of megaflows can be enhanced and the netlink overhead of OVS can be significantly reduced.

6 Evaluation

The performance of HBD is evaluated through simulation, and three other algorithms are implemented for comparison, which includes: (1) the naive exact dispatching approach; (2) a dependency-based algorithm that works in reactive mode, copies the strategy of Cover-set[12], and treats the hits on the cover sets as cache misses (We refer to it as Dependency-Based Dispatching, DBD); and (3) the cache rule generation algorithm used in CAB[16].

For DBD, it should be noted that in the experiments, we use the Cover-set strategy as a benchmark on the reactive dispatching model, which is different from the use case in CacheFlow (i.e., in the proactive dispatching model)[12]. If the number of dependent rules triggered by a packet-in exceeds the cache limit, DBD simply dispatches an exact-match rule.

6.1 Data set and test-bed

The rule sets used for the evaluation include: (1) two real-life rule sets described in Section 4, vAccess and vGateway; and (2) another two larger-sized synthetic rule sets in Classbench[19], ACL1-5K and FW1-5K, with 4415 and 4653 five-tuple rules, respectively.

For vAccess, the corresponding traces are captured at the physical server where the access software switch locates, with a duration of one hour. For vGateway, the traces are captured at the virtual gateway with a duration of 15 minutes. For the two synthetic rule sets, we rewrite the trace generation function of ClassBench to generate synthetic traces that better follow the distribution of real-life traffic (in terms of locality). Concretely, we keep Pareto distribution used in trace generator algorithm the same and replace the copy function, for depicting the temporal locality of packets. Besides, we introduce Zipf distribution to replace the random-corner function, for better depicting the spacial locality of flows.

On an HP Z220 SFF workstation with 3.40 GHz CPU, 16 GB memory, and 64 bit Ubuntu Server 12.04 LTS, we simulate a controller along with a connected forwarding device. In the controller, the HyperSplit[13] algorithm is adopted for finding the matching rule of a "cache miss" packet. In the forwarding device, the Least Recently Used (LRU) algorithm is adopted for cache rule replacement. Besides, all the source code of the evaluated rule dispatching algorithms is written in C language.

6.2 Experimental results

For the optimization objective of (1) reducing $T_{\text{avrg}} = T_{\text{hit}} + \eta_{\text{miss}} \times (T_{\text{trans}} + T_{\text{gen}})$ and (2) satisfying the rule space constraints in the forwarding devices, we first define the evaluation metrics.

- **Cache miss ratio (η_{miss})**: the percentage of cache miss packets when testing on the corresponding trace of a specific rule set.
- **Average cache generation time (T_{gen})**: the maximum packet-in rate that guarantees no packet loss in the controller dispatcher.
- **Average Traffic Coverage Capability (ATCC)**:

$$\text{ATCC} = \frac{\text{hyper\_volume}(\cup R_{\text{cache}})}{n} \tag{8}$$

where n is the maxinum rule number of a forwarding device, $\cup R_{\text{cache}}$ is the union of all hyper-rectangles representing the current cache rules. Note that this metric reflects the coverage range of cache rules but does not regard traffic locality.

- **Effective ATCC (e-ATCC):**

$$\text{ATCC} = \frac{\text{hyper\_volume}(\cup R_{\text{effective-cache}})}{n} \qquad (9)$$

where $R_{\text{effective-cache}}$ is the cache rules that are actually hit by traffic.

6.2.1 Effectiveness on reducing T_{avrg}

The cache miss ratios η_{miss} across a range of rule-cache size are illustrated in Fig. 7. As a benchmark, the exact dispatching method reflects the temporal locality of traffic. According to the results, though DBD better reduces the cache miss ratio against the exact dispatching method, the cache capability achieved by CAB and HBD are much more significant, indicating the feasibility and benifits of reactive rule dispatching. For example, on the rule set vAccess, cache rules generated by HBD are able to cover 90% traffic with only 8% cache size. Compared with CAB, we can see that the smaller the cache size, the better cache capability HBD achieves than CAB. It is due to the "one cache rule per packet-in" principle of HBD, which concentrates on making optimal use of every single rule space in the forwarding devices.

For calculating T_{gen}, we measure the throughput of the rule dispatchers, i.e., the maximum packet-in rate that guarantees no packet loss in the dispatcher. As shown in Table 4, the throughput of the HBD dispatcher is around one million of packets per second (pps), indicating that $T_{\text{gen}} \ll T_{\text{trans}}$. Thus, according to the analysis in Section 3, HBD guarantees that the benefit (i.e., the significant reduction of η_{miss}) outweighs the cost (i.e., T_{gen}).

Table 4 HBD dispatcher throughput.

Set	Throughput (Mpps)
ACL1_5K	1.19
FW1_5K	0.82
vAccess	1.55
vGateway	0.98

6.2.2 Efficiency on memory utilization of forwarding device

The ATCC and the e-ATCC are used for testing device memory utilization. Unlike the ATCC calculating all the dispatched cache rules, the e-ATCC only takes into consideration the effective cache rules among them (i.e., rules that are actually hit by the traffic). For example, the cache rules dispatched by DBD have high ATCC, but only a small proportion of them are actually useful for handling traffic, which is indicated by e-ATCC. For HBD, ATCC equals to e-ATCC because of the "one cache rule per packet-in" principle.

To calculate the ATCC and the e-ATCC for a specific algorithm, we set the rule-cache size to 10% (a reasonable cache size with both space efficiency and cache hit rate guaranteed), and take three cache snapshots during the algorithm processing. The ATCC and the e-ATCC illustrated in Fig. 8 are the average value in the three snapshots. As we can see, though the ATCC of HBD is usually less than that of DBD and CAB (because HBD calculates the accurate "inscribed"

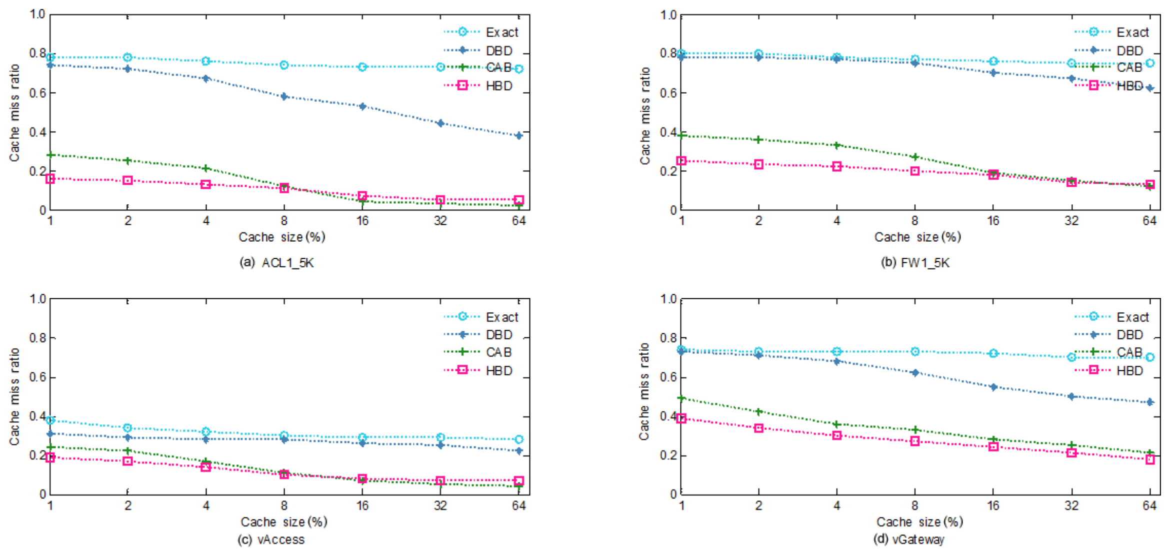


Fig. 7 Cache miss ratio.

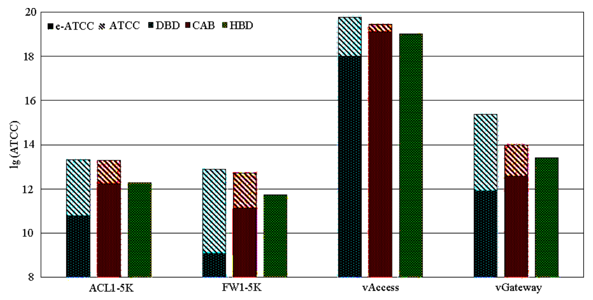


Fig. 8 Traffic coverage capability.

hyper-rectangle representing a cache rule, instead of finding several intact rules or generating a large-coverage bucket), the e-ATCC of HBD is usually larger than that of DBD and CAB, especially for the rule sets with serious rule overlaps such as FW1-5K and vGateway.

6.2.3 Effectiveness of the greedy cache generation algorithm

To test the effectiveness of the greedy cache rule generation algorithm, we compare the cache rules generated by HBD with the cache rules generated by the optimal solution presented in Section 3. As shown in Table 5, on the two small-sized real-life rule sets, ACL1 and FW1, ATCC of the HBD-generated rules is close to ATCC of the optimal rules. The latter takes our workstation more than twenty hours to compute.

To conclude, according to the above experimental results, the naive exact dispatching approach suffers from high cache miss penalty due to the weak traffic coverage capability of exact-match rules; and the dependency-based algorithm is not suitable for reactive dispatching mode because it usually inevitably cache numerous dependent rules (even if the cover-sets splice the dependency chains), most of which are ineffective for covering upcoming traffic. Given limited cache size, HBD outperforms CAB in terms of cache hit ratio. Addtionally, to achieve the nice performance, CAB requires the underlying forwarding devices to support two-stage flow table pipeline. In contrast, HBD is compatible with the legacy forwarding devices with any kind of packet classification approach applied. Moreover, as an additional (but not mandatory) bonus option, the forwarding devices can leverage the order-independent attributes of cache rules dispatched by HBD to further optimize their rule lookup processes.

Table 5 ATCC: Greedy vs. optimal.

Method	ATCC on ACL1	ATCC on FW1
Greedy	15.25	14.85
Optimal	16.40	15.50

7 Conclusion

In this work, we take a deep-dive to the reactive rule dispatching problem in software-defined networks, and propose the HBD approach for optimizing the rules to be dispatched. Compared with existing approaches, the strategies used in HBD enable the network to achieve better performance in terms of transmission latency and device memory utilization.

The code of HBD and the trace generation function we used for evaluation will be available on our website[26] to encourage more intensive research in this area.

References

[1] ONF Market Education Committee, Software-defined networking: The new norm for networks, ONF White Paper, 2012.

[2] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, Scalable flow-based networking with DIFANE, *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 351–362, 2011.

[3] M. Moshref, M. Yu, A. Sharma, and R. Govindanm, vcrib: Virtualized rule management in the cloud, in *Proc. NSDI*, 2013.

[4] Y. Kanizo, D. Hay, and I. Keslassy, Palette: Distributing tables in software-defined networks, in *Proc. INFOCOM*, 2013, pp. 545–549.

[5] N. Kang, Z. Liu, J. Rexford, and D. Walker, Optimizing the one big switch abstraction in software-defined networks, in *Proc. CoNEXT*, 2013, pp. 13–24.

[6] K. C. Claffy, Internet traffic characterization, PhD dissertation, UCSD, CA, USA, 1994.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, Rethinking enterprise network control, *IEEE/ACM Transactions on Networking*, vol. 17, no. 4, 1270–1283, 2009.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, et al., Onix: A distributed control platform for large-scale production networks, in *Proc. OSDI*, 2010, vol. 10, pp. 1–6.

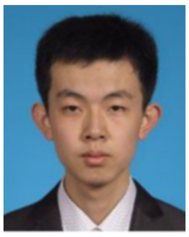
[9] Y. Hassas and Y. Ganjali, Kandoo: A framework for efficient and scalable offloading of control applications, in *Proc. HotSDN*, 2012, pp. 19–24.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, DevoFlow: Scaling flow management for high-performance networks, *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

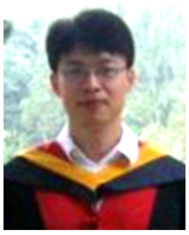
[11] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, SAX-PAC (scalable and expressive packet classification), in *Proc. SIGCOMM*, 2014, pp. 15–26.

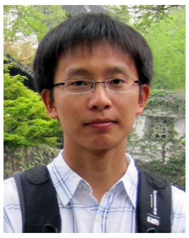
[12] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, Infinite cacheflow in software-defined networks, in *Proc. HotSDN*, 2014, pp. 175–180.
[13] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, Packet classification algorithms: From theory to practice, in *Proc. INFOCOM*, 2009, pp. 648–656.
[14] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, Wire speed packet classification without tcams: A few more registers (and a bit of logic) are enough, *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1, pp. 253–264, 2007.
[15] Y. Ma, S. Banerjee, S. Lu, and C. Estan, Leveraging parallelism for multi-dimensional packet classification on software routers, *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, 227–238, 2010.
[16] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, CAB: A reactive wildcard rule caching system for software-defined networks, in *Proc. HotSDN*, 2014, pp. 163–168.
[17] P. Gupta and N. McKeown, Packet classification using hierarchical intelligent cuttings, in *Proc. Hot Interconnects VII*, 1999, pp. 34–41.
[18] S. Singh, F. Baboescu, G. Varghese, and J. Wang, Packet classification using multidimensional cutting, in *Proc. SIGCOMM*, 2003, pp. 213–224.
[19] Evaluation of packet classification algorithms, http://www.arl.wustl.edu/ hs1/PClassEval.html, 2015.
[20] X. Wang, Y. Qi, Z. Liu, and J. Li, LiveCloud: A lucid orchestrator for cloud datacenters, in *Proc. CloudCom*, 2012, pp. 341–348.
[21] X. Wang, Z. Liu, B. Yang, Y. Qi, and J. Li, Tualatin: Towards network security service provision in cloud datacenters, in *Proc. ICCCN*, 2014, pp. 1–8.
[22] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, Conflict classification and analysis of distributed firewall policies, *Selected Areas in Communications, IEEE Journal on*, vol. 23, no. 10, pp. 2069–2084, 2006.
[23] B. Vamanan, G. Voskuilen, and T. Vijaykumar, Efficuts: Optimizing packet classification for memory and throughput, *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 207–218, 2010.
[24] X. Wang, C. Chen, and J. Li, Replication free rule grouping for packet classification, *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 539–540, 2013.
[25] Open vSwitch 2.3.1, http://openvswitch.org/, 2015.
[26] NSLab, http://security.riit.tsinghua.edu.cn/, 2015.

Chang Chen received the BEng degree from Tsinghua University, China, in 2015. He is now a master student in Department of Automation at Tsinghua University. His research interests include software-defined networking, high-performance algorithms in computer networking, and system architectures.


Xiaohe Hu received the BEng dgree from Tsinghua University, China, in 2014. He is now a PhD candidate in Department of Automation at Tsinghua University. His research interests include software-defined networking and cloud datacenter networks.

Jun Li received the BEng amd MEng degrees from Tsinghua University, China, in 1985 and 1988, respectively, and the PhD degree from New Jersey Institute of Technology, in 1997. He is currently a professor and the dean at the Research Institute of Information Technology, and the executive vice dean of the School of Information Science and Technology, Tsinghua University. He was a board director of XML Global and is currently a board member or advisor to several venture capital and startup companies, including GigaDevice and Agate Logic. He is also a deputy director of the Tsinghua National Lab for Information Science and Technology. He is a co-author of more than 100 papers, and co-inventor of 10 patents. He is also a managing director of an angle fund Versatile Venture Capital. His research interests include network security, pattern recognition, and image processing.


Kai Zheng received the BS degree from Beijing University of Posts and Telecommunications, China, in 2001, and the PhD degree from Tsinghua University, China, in 2006. From then on he joined IBM China Research Lab as a research staff member on computer system architecture. His research interests include cloud networking, software-defined networking, and software-defined protocol.


Xiang Wang received the BS degree from Xidian University, China, in 2007, and the MS degree from University of Science and Technology of China, China, in 2010. He is working toward the PhD degree at the Department of Automation, Tsinghua University, China. His research interests include software-defined networking, distributed system, and performance issues in computer networking, and system architectures.

Yang Xiang received the BS degree from Jilin University, China, in 2008, and the PhD degree from Tsinghua University, China, in 2013. He is currently a postdoctor in the Network Security Lab, Research Institute of Information Technology, Tsinghua University. His research interests include software defined network, network architecture, inter-domain routing security, and intrusion detection.