TCS-TR-A-14-74

TCS Technical Report

Packet Classification for Global Network View of Software-Defined Networking

by

Takeru Inoue, Toru Mano, Kimihiro Mizutani,
Shin-ichi Minato, and Osamu Akashi

Division of Computer Science

Report Series A

July 16, 2014

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682
Fax: +81-011-706-7682

Packet Classification for Global Network View of Software-Defined Networking

Takeru Inoue*† Toru Mano† Kimihiro Mizutani†
Shin-ichi Minato‡ Osamu Akashi†

July 16, 2014

Abstract

In software-defined networking, applications are allowed to access a global view of the network so as to provide sophisticated functionalities, such as quality-oriented service delivery, automatic fault localization, and network verification. All of these functionalities commonly rely on a well-studied technology, packet classification. Unlike the conventional classification problem to search for the action taken at a *single switch*, the global network view requires to identify the *network-wide* behavior of the packet, which is defined as a combination of switch actions. Conventional classification methods, however, fail to well support network-wide behaviors, since the search space is complicatedly partitioned due to the combinations.

This paper proposes a novel packet classification method that efficiently supports network-wide packet behaviors. Our method utilizes a compressed data structure named the multi-valued decision diagram, allowing it to manipulate the complex search space with several algorithms. Through detailed analysis, we optimize the classification performance as well as the construction of decision diagrams. Experiments with real network datasets show that our method identifies the packet behavior at 20.1 Mpps on a single CPU core with only 8.4 MB memory; by contrast, conventional methods failed to work even with 16 GB memory. We believe that our method is essential for realizing advanced applications that can fully leverage the potential of software-defined networking.

1 Introduction

Software-Defined Networking (SDN) [1] is a new paradigm that separates the control and forwarding planes in a network. The control plane, which is operated

*takeru.inoue@ieee.org
†NTT Network Innovation Laboratories, Yokosuka, Japan.
‡Graduate School of Information Science and Technology, Hokkaido University, Sapporo, Japan.

by a logically centralized controller, provides a global view of the distributed network state. This global view allows SDN applications running on the control plane to readily identify *network-wide packet behaviors*, e.g., which path the packet traverses, whether the packet is discarded in the network, and what actions the packet is subjected to at middle boxes. Many applications have been developed that utilize the network-wide packet behavior to offer sophisticated functionalities.

- In quality-oriented services [2] and an upcoming infrastructural paradigm called "network fabric" (separation of intelligence from the network core) [3, 4], the network edge is assumed able to determine the network-wide behavior for every incoming packet.
- Reference [5] proposed an automatic fault localization technique that exploits the differences between actual packet behavior monitored in the network and the theoretical behavior estimated from the global view.
- References [6–11] estimate the network-wide packet behavior from the network status, in order to verify conformity with a network policy.

An method that can efficiently determine the network-wide behavior of a packet is essential to successfully implementing these advanced applications; they cannot work if only the action to be taken at a *single* switch can be identified.

1.1 Summary and Limitations of Prior Art

Packet classification [12–20], a functionality that determines the action taken on a packet based on multiple header fields, has been a key technology in modern networks to provide services beyond basic packet forwarding, such as access control, quality of services, and traffic monitoring and analysis. Packet classification has been extensively studied in the past fifteen years, and the state-of-the-art method [16] looks up large classifiers with tens of thousands of rules very efficiently. These well-studied methods are, however, easily overwhelmed by the complexity of handling network-wide behaviors. This is because, as shown in Fig. 1, network-wide behavior is defined as the *combinations* of switch actions. Our preliminary experiments on the Stanford backbone network [10], which is a medium-scale network with 16 switches, 757,170 forwarding rules, and 1,584 ACL rules, revealed that the classification method of [16] fails to construct a classifier of the network-wide behaviors, exhausting the available computer memory of 16 GB. The main cause of this failure is that the search space defined by the packet header was partitioned into a huge number of blocks; 652 *million* rules were required to express the complicated space, but the conventional methods baulk at this number.

In order to efficiently handle such a complicated partitioned space, some network verifiers [6–8] employ Binary Decision Diagrams (BDDs) [21], which are a compressed data structure designed to represent Boolean functions. Due to the great space efficiency of BDDs, only 6.6 MB memory was required to represent

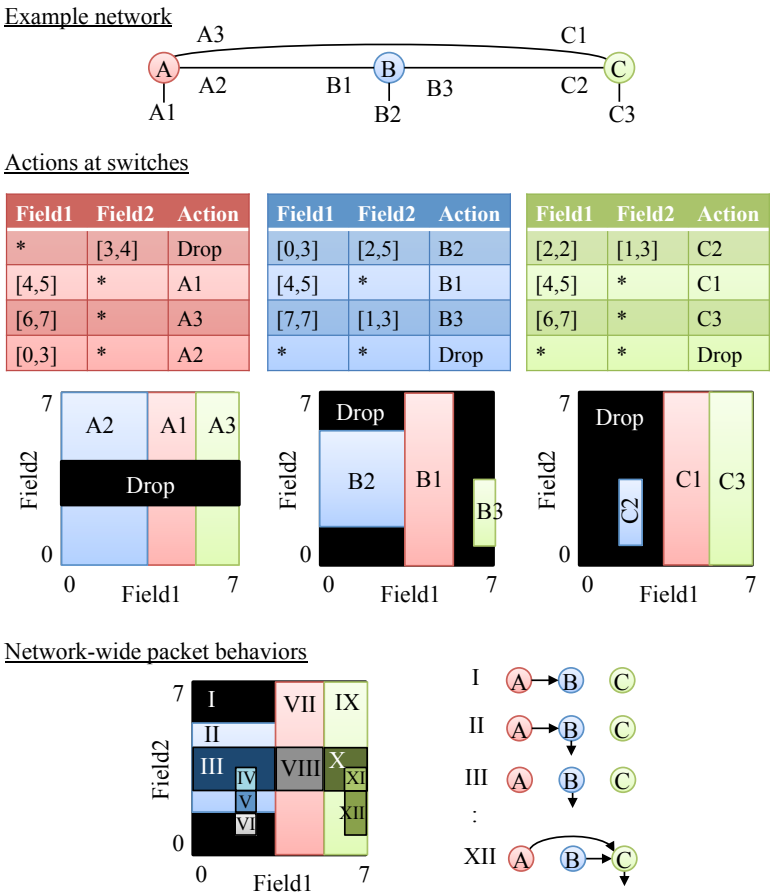

Figure 1: Search space defined by packet header and network-wide packet behaviors. The example network at the top includes three switches (A, B, and C), each of which has three ports. Each switch has a table that maintains several rules, and each rule associates two header fields (field1 and field2 of 3-bit) with actions. The two-field header space (a square) is divided into only seven blocks at switch A, but it is partitioned into 17 blocks for the twelve network-wide behaviors as shown in the bottom.

the Stanford backbone in our experiments. However, a behavior must be looked up by linear search which is impractically slow; considering the several Boolean functions represented in BDDs, each of which maps the header space exclusively to a network-wide packet behavior, these functions must be tested individually to determine the packet behavior (e.g., in Fig. 1, twelve Boolean functions have to be tested one by one). In our experiments on the Stanford backbone, which has 1,093 network-wide behaviors, only 5.78 Kpps was achieved with BDDs. Classification throughput should exceed 10 Mpps in order to examine packets at line rate on 10 Gbps links.

As discussed so far, no method has been introduced that can identify network-wide packet behavior with the necessary time and space efficiency. This deficiency imposes the following limitations on the SDN applications discussed earlier on.

- Quality-oriented services and network fabric cannot be realized in a complex network like the Stanford backbone, whose complicated network-wide packet behaviors cannot be maintained by network edge switches using conventional classification methods.
- Reference [5] only employs predefined test packets for failure detection, since it is hard for conventional methods to identify the theoretical network-wide behavior for arbitrary packets; this restriction might hinder the resolution of faulty behaviors of production traffic.
- References [6–11] can test only a few properties every few msec, which may cause serious policy violations to be overlooked if a lot of properties have to be checked in real-time.

Similar limitations are faced by other applications. In addition, the limitations will hinder the advent of more advanced SDN applications. To remove the limitations and leverage the full potential of SDN, it is imperative to develop a new classification method satisfying the following requirements; the complicated header space of network-wide packet behaviors should be represented compactly enough to fit within a fast cache memory, and packets at the rate of beyond 10 Mpps should be classified based on their network-wide behavior. Rapid construction and update of classifiers are also appreciated.

1.2 Our Contributions

This paper proposes a novel packet classification method that meets the above requirements. Our idea is simple but very effective. Boolean functions, each of which is associated with a single behavior, are unified into a single *multi-valued function* representing a whole classifier by itself. This multi-valued function is represented by a data structure named the *multi-valued decision diagram* (MDD) [22], a variant of BDD. Since this unification removes the slow linear search from the classification process, the throughput can be greatly improved while maintaining the space-efficiency of BDDs.

Our method is two-fold.

- An efficient algorithm that constructs the MDD from a set of BDDs representing Boolean functions, is presented. The construction process is analyzed and optimized by reducing it to Huffman coding. Upon receipt of a new packet behavior, the MDD can be incrementally updated in a short time.
- A very simple and fast algorithm examines the MDD to classify packets based on their behaviors. Another algorithm further accelerates the classification process by regarding a bunch of header bits as a single variable. The time-space tradeoff entailed in the bit aggregation is analyzed.

Our method is evaluated with three real network configurations: Internet2[1], Stanford backbone network [10], and Purdue campus network [23]. In the case of Stanford, the MDD only requires 8.4 MB of memory, which fits within fast CPU cache. The classification throughput is 20.1 Mpps on a single CPU core, nearly 3,500 times faster than conventional BDD methods. The MDD is constructed in 0.93 sec, and a new behavior can be added just in 29 msec.

Our method does not require any special hardware support like TCAMs, and can be applied to any use case without significant difficulties. In this paper, our method is implemented as a fully software-based classifier assuming its use in software-based SDN applications, but it is so simple that it can be realized as a hardware-based system.

The rest of this paper is organized as follows. After defining our problem in Section 2, Section 3 reveals deficits of conventional methods. Section 4 designs our method, and Section 5 elaborates algorithms used in the method. Section 6 reports the experiments and their results, and Section 7 discusses application aspects. Section 8 summarizes related work, and Section 9 concludes this paper.

2 Problem Statement

The logical search space defined by the L-bit packet header is denoted by $\mathcal{X} = \{0,1\}^L$, and is called the *header space* [10] (e.g., a square in Fig. 1 represents a header space of $L = 3 + 3$). Packet header x corresponds to a point in the header space, $x \in \mathcal{X}$. The protocol-specific semantics of the packet header are ignored in our method, and a packet header is considered as a flat sequence of bits.

The header space is partitioned into *equivalent subspaces*, each of which corresponds exclusively to a single behavior (e.g., the bottom square of Fig. 1 has twelve equivalent subspaces). An equivalent subspace is not necessarily continuous, and can consist of several blocks, where *block* is used to specify a continuous subspace. Let X_i be an equivalent subspace that maps to the i-th behavior, the subspace is defined as,

$$X_i = \{x \in \mathcal{X} : f_i(x) = \top\},$$

where $f_i(x)$ is a Boolean function indicating whether a packet with header x follows the i-th behavior, $f_i : \mathcal{X} \to \{\bot, \top\}$, and $\bot$ is false and $\top$ is true. Since equivalent subspaces are mutually exclusive, $X_i \cap X_j = \emptyset$ $(i \neq j)$, and collectively exhaustive, $\bigcup_i X_i = \mathcal{X}$, they form a partition of header space $\mathcal{X}$, P, as follows,

$$P = \{X_{\mathrm{I}}, X_{\mathrm{II}}, \cdots, X_{|P|}\}.$$

The equivalent subspaces, X_i's, and corresponding Boolean functions, f_i's, are obtained by utilizing switch actions as follows [8]. Given P_1 and P_2 as arbitrary partitions of $\mathcal{X}$, we define operation $\otimes$ as,

$$P_1 \otimes P_2 = \{X \cap Y : X \in P_1, Y \in P_2, X \cap Y \neq \emptyset\}.$$

[1] `http://vn.grnoc.iu.edu/Internet2/fib/`

Using this operation, the whole header space is then partitioned into equivalent subspaces as follows,

$$P = \bigotimes_j P_j, \tag{1}$$

where P_j is a partition defined by the actions of the j-th switch (e.g., a square in the middle of Fig. 1)[2]. This operation divides the header space into several blocks, which become combinatorially finer (e.g., the bottom square is much finer than middle squares in Fig. 1). In other words, intersecting rules of different switches spawn new blocks to distinguish network-wide behaviors, while rules on a single switch are simply masked by higher-priority rules if intersected.

Multi-valued function F is defined by,

$$F : \mathcal{X} \to \{\text{nil}, \text{I}, \text{II}, \cdots, |P|\},$$

where $F(x) = i$ if and only if header x is in the i-th equivalent subspace, $x \in X_i$. A multi-valued function that cannot be "nil" is called a *complete* multi-valued function, and can be used as a packet classifier.

Our goal in this paper is to construct an efficient data structure representing a complete multi-valued function, F, given a set of Boolean functions, f_i's.

3 Inefficiencies of Conventional Packet Classification Methods

This section identifies why conventional packet classification methods fail to support network-wide packet behaviors. There are two major conventional approaches; algorithmic methods [12–16] usually construct a decision tree to represent the header space, while architectural methods [17–20] often try to reduce the number of rules to fit them in space-limited TCAMs. These conventional methods commonly depend on the following assumption; the header space is a multi-dimensional space defined by each header field (not each bit like our definition), and it is covered by a set of *hypercubes*, where a hypercube is a convex block defined by a range or prefix in each field. To resolve possible conflicts among hypercubes, they are given an overall total order. Normally, hypercubes are specified by a list of "5-tuple" rules (i.e., source/destination IP, source/destination port, and protocol).

The computational complexity of both conventional approaches depends on the number of hypercubes, as follows.

- In a decision tree, the root node represents the whole header space, which is recursively divided at every intermediate node until a few rules are left

[2]Note that these partitions, P_j's, might be given for each port as well as each switch, depending on network configuration.

at a leaf node. The space complexity is known as $O(N^D)$ for $O(\log N)$ classification time [24], where N is the number of hypercubes and D is the number of header fields. The required space in practice could be smaller than this worst-case bound, but it still depends on the number of hypercube rules because the decision tree must be large enough to divide the rule set into several subsets with a few rules.

- Rule reduction methods reduce the number of hypercube rules by merging those of the same action into a single large hypercube; in the following example, the left set of hypercube rules is reduced to the right set.

IP destination	Action
0.0.0.0/3	Drop
32.0.0.0/3	A1
64.0.0.0/2	Drop
128.0.0.0/1	Drop

IP destination	Action
32.0.0.0/3	A1
0.0.0.0/0	Drop

The time complexity of TCAM razor, the most-cited rule reduction method, is roughly considered to be $O(DNAL^2)$, where A is the number of actions/behaviors, since it performs dynamic programming of $O(NAL^2)$ [25] for each header field.

The number of hypercubes, N, is examined for the three real networks (the statistics are given in Section 6). To our knowledge, there is no technique that can calculate a set of hypercubes to represent a header space of network-wide behavior, and so hypercubes are extracted from BDDs of f_i's.

	Internet2	Stanford	Purdue
# of hypercubes	35,700	652,115,821	834,648,394

The numbers for Stanford and Purdue networks are extremely large. This is because Stanford and Purdue include multi-field rules, which incurs the curse of dimensionality [26]. Internet2 dataset includes single-field rules only, and so it has a moderate number of hypercubes.

Against these networks, we evaluate three conventional methods: two decision tree methods, HyperCuts [12] and HybridCuts [16], and one rule reduction method, TCAM razor [18]. Open-source implementations were available for HyperCuts[3] and HybridCuts[4], while our own implementation was developed for TCAM razor. As shown in the following table, the single success was achieved by HybridCuts for Internet2; otherwise, the conventional methods exhausted 16 GB memory or did not finish in a half hour.

	Internet2	Stanford	Purdue
HyperCuts	fail (memory)	fail (memory)	fail (memory)
HybridCuts	success	fail (memory)	fail (memory)
TCAM razor	fail (timeout)	fail (timeout)	fail (timeout)

[3] `http://hypercuts.masicek.net/`, by Charles University.
[4] `http://github.com/lwj4333765/HybridCuts`, by the authors of HybridCuts.

To recap, although the header space of network-wide behavior requires a huge number of hypercube rules to represent it, the computational complexity of conventional methods is directly dependent on the number. In order to remove this dependency, our method should be constructed from a compressed representation like BDDs where individual rules do not need to be examined.

4 Abstract Procedure of Proposed Method

This section describes the abstract procedure of our method; algorithms to implement the procedure are given in Section 5.

Our method, first, builds incomplete multi-valued function F_i from the Boolean function of the i-th behavior, f_i. Function F_i maps the header space to just the i-th behavior, as follows,

$$F_i : \mathcal{X} \to \{\text{nil}, i\},$$

where $F_i(x) = i$ if $f_i(x) = \top$, or $F_i(x) = \text{nil}$ otherwise.

Since two equivalent subspaces, X_i and X_j $(i \neq j)$, are mutually exclusive, two incomplete multi-valued functions, F_i and F_j, never define behaviors for the same packet header; i.e., $\forall x \in \mathcal{X}, F_i(x) = \text{nil} \vee F_j(x) = \text{nil}$. These two incomplete multi-valued functions can be unified into a single one without conflict, by introducing the following unification operation,

$$F_i(x) \uplus F_j(x) = \begin{cases} F_i(x) & \text{if } F_j(x) = \text{nil}, \\ F_j(x) & \text{if } F_i(x) = \text{nil}, \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2}$$

This operation is clearly associative and commutative.

Performing this operation over multi-valued functions of all the behaviors yields the multi-valued function of classifier, $F(x)$, as follows,

$$F(x) = \biguplus_{i=\mathrm{I}}^{|P|} F_i(x). \tag{3}$$

Since the equivalent subspaces are collectively exhaustive, $F(x)$ is complete, that is, $\forall x \in \mathcal{X}, F(x) \neq \text{nil}$.

To incrementally update classifier F, we consider that rules of another multi-valued function, F', would be inserted above the original rules. More formally, $F(x)$ is overwritten by $F'(x)$ just in the subspace X' in which $F'(x)$ is defined, $\forall x \in X', F'(x) \neq \text{nil}$. This update operation is defined as,

$$F'(x) \lhd F(x) = \begin{cases} F'(x) & \text{if } F'(x) \neq \text{nil}, \\ F(x) & \text{otherwise.} \end{cases} \tag{4}$$

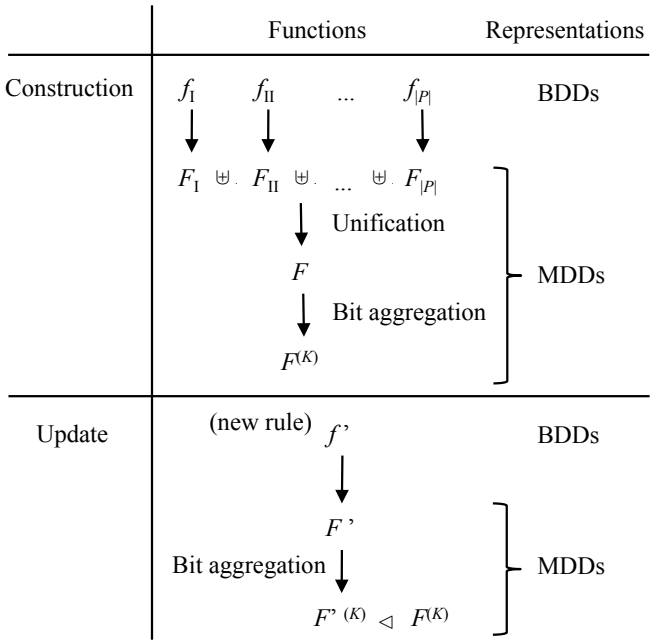

Figure 2: Flowchart of classifier construction and update in our method.

Note that update operation $\lhd$ is equivalent to unification operation $\uplus$ when the latter operation is defined, but this paper distinguishes them for clarity.

In order to accelerate the packet classification process, a multi-valued function in which continuous K bits are aggregated into a single variable is defined as follows,

$$F^{(K)} : \{0, 1, \cdots, 2^K - 1\}^{\frac{L}{K}} \rightarrow \{\mathrm{nil}, \mathrm{I}, \mathrm{II}, \cdots, |P|\}.$$

Necessarily, $F^{(K)}(x) = F(x), \forall x \in \mathcal{X}$. This paper uses $F^{(K)}$ only when it has to be specified by K, or uses simply F otherwise.

Last, the construction and update procedures of our method are summarized as a flowchart in Fig. 2.

- In classifier construction, Boolean functions mapped to behaviors, f_i's, are converted to multi-valued functions, F_i's, which are unified into the multi-valued function of classifier, F, and further converted to $F^{(K)}$ by bit aggregation.
- To incrementally update classifier $F^{(K)}$, new rule f' is converted into $F'^{(K)}$ through bit aggregation, and then inserted to the classifier.

5 Algorithms in Proposed Method

Given a set of BDDs representing Boolean functions, f_i's, which are obtained using existing techniques like [8], the algorithms proposed in this section construct

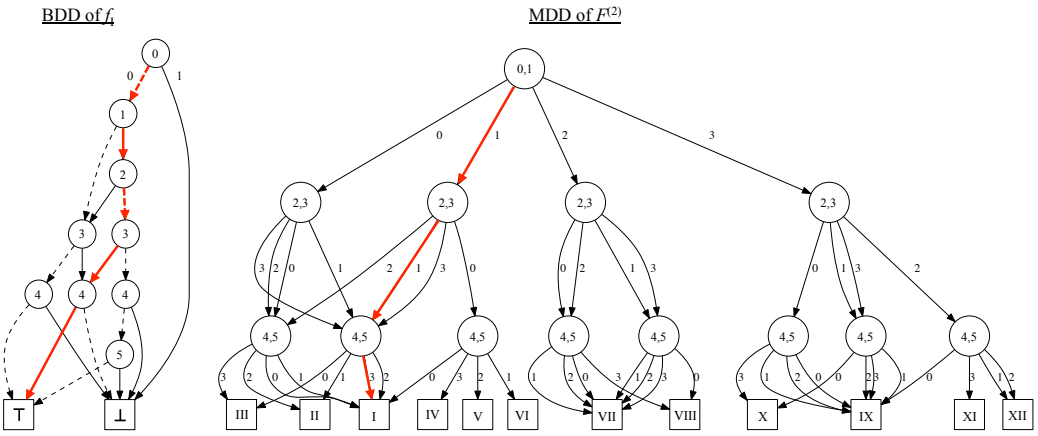

Figure 3: BDD (left) and MDD of $K = 2$ (right). The BDD represents the Boolean function mapped to behavior "I" of Fig. 1, while the MDD represents the multi-valued function mapped to all twelve behaviors of Fig. 1. Since consecutive two-bits are aggregated in the MDD, non-terminal nodes are labeled by the two-bits while arcs are labeled by 0- to 3-child. A packet header of $x = 010111$ is indicated by the red paths in the both diagrams.

the MDD of $F^{(K)}$. After reviewing decision diagrams in Section 5.1, Section 5.2 analyzes the unification and update operations performed on MDDs. Section 5.3 introduces the bit aggregation algorithm, and Section 5.4 proposes a search algorithm for the packet classification.

5.1 Decision Diagrams

As shown in Fig. 3 (left), a BDD [21] is an acyclic directed graph with a single root node and two terminal nodes, $\bot$ and $\top$. Each non-terminal node is labeled as b-th bit in the header, $b \in [0, L-1]$, and it has two labeled arcs, 0-child and 1-child, each of which indicates the b-th bit is 0 or 1. A path from the root to a terminal corresponds to a packet header, x, or a set of headers (a hypercube) when some bits are skipped (e.g., the red BDD path in Fig. 3, on which the fifth bit is skipped, expresses a hypercube of $01011*$). The terminal node at the end of path indicates the value of $f(x)$. The maximum height, or the longest path, is L. Common prefix and suffix are shared among paths for compression (e.g., two paths, $00*11*$ and $01111*$, share their prefix 0 and suffix $11*$ in Fig. 3); it is believed that BDDs are well compressed for most practical functions [27]. The size of BDD is defined as the number of non-terminal nodes in it, and the size is denoted by $||f||$ if the BDD represents function f.

An MDD [22], which is shown in Fig. 3 (right), is also an acyclic directed graph with a single root, but it can have more than two terminal nodes, $\mathrm{nil}, \mathrm{I}, \mathrm{II}, \cdots, |P|$. Each non-terminal node is labeled by aggregated bits, and it can have more than two children arcs, $0, 1, \cdots, 2^K - 1$. The maximum height is L/K. Other properties are the same as those of BDD.

In our method, a BDD is used to represent a Boolean function that maps the header space to a single packet behavior, while an MDD is used to express a multi-valued function. BDD of $f_i(x)$ is easily converted to MDD of $F_i(x)$, by replacing $\bot$- and $\top$-terminals with nil- and i-terminals, respectively. The time complexity of this operation is clearly constant, $O(1)$. MDD size is obviously the same as BDD size, i.e., $||F_i|| = ||f_i||$.

5.2 Unification and Update Operations

5.2.1 CASE Algorithm

Reference [22] defines an efficient algorithm named CASE that performs arbitrary operations over two operand MDDs and constructs the resulting MDD. This CASE algorithm allows our method to apply operations $\uplus$ and $\lhd$ to multi-valued functions so as to calculate (2) and (4).

The worst-case MDD size can be quite large, $||A \diamond B|| \leq ||A|| \cdot ||B||$, where A and B are multi-valued functions and $\diamond$ is an arbitrary operator. However, the size is usually considerably smaller than this worst-case upper bound, closer to $||A|| + ||B||$ [28]; in our experiments, the size was never greater than the total size of operands, though impractical counter examples can be considered. This observation yields the following assumption.

Assumption 1. *The size of MDD performing a single operation is less than or equal to the total size of operand MDDs,*

$$||A \diamond B|| \leq ||A|| + ||B||. \tag{5}$$

The time complexity of CASE algorithm is defined by the number of nodes and that of children per node [22], and so it is fixed with Assumption 1 as follows,

$$O(2^K(||A|| + ||B||)), \tag{6}$$

where the algorithm involves $||A|| + ||B||$ node creations, at each of which 2^K children are set. Note that A and B must share a common K to apply CASE algorithm.

5.2.2 Unification

The MDD size and time complexity of (3) is analyzed. Using (5), the MDD size is bounded as follows,

$$||F|| \leq \sum_{i=\mathrm{I}}^{|P|} ||F_i||. \tag{7}$$

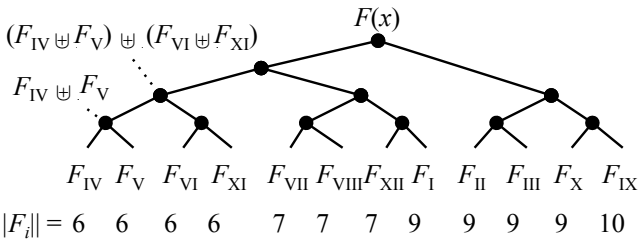

Figure 4: Huffman tree representing the optimal calculation order of (3) for Fig. 1. Leaf nodes are MDDs of F_i's, while internal nodes represent $\uplus$ operations. The MDD sizes, $||F_i||$'s, are shown at the bottom.

Since operation $\uplus$ is associative and commutative, operations in (3) can be performed in an arbitrary order. In addition, the time complexity of each operation in (3) varies depending on the size of operand MDDs. We can, therefore, optimize the calculation order.

Theorem 1. *The time complexity of (3) optimized with Huffman coding is given by,*

$$O\left(2^K \sum_{i=\mathrm{I}}^{|P|} \ell(F_i)||F_i||\right), \tag{8}$$

where $\ell(F_i)$ is the path length from the root to leaf F_i on the Huffman tree.

Proof. Since each operation takes only two operands at a time in the CASE algorithm, the calculation order can be illustrated as a binary tree, like Fig. 4. Every function F_i on a leaf is used at each internal node up to the root, and so the time complexity of (3) is bounded by (8) using (6).

The optimal calculation order is given as a binary tree minimizing (8); this optimization can be regarded as Huffman coding [29], as leaf MDDs and their sizes are replaced with symbols and their weights (frequencies), respectively. This binary tree is also known as a Huffman tree. Note that the time complexity ignores Huffman tree construction, since it is negligible compared with MDD operations. □

5.3 Bit Aggregation

This subsection defines the bit aggregation algorithm that accelerates packet classification. Algorithm 1 constructs the MDD of $F^{(K)}$ from that of F, by aggregating continuous K bits into a single variable. Note that in Algorithm 1, F or $F^{(K)}$ refers to the root node of MDD representing function F or $F^{(K)}$, not to the function itself. This algorithm recursively creates MDD nodes by finding their 2^K children at K bits ahead from the node; x' means K continuous bits from $F.b$, where $F.b$

Algorithm 1: Aggregate

```
Input: F
Output: F^(K)
if F is non-terminal or F not found in cache then
    create F^(K)
    for x' ← 0 to 2^K − 1 do                    // x' is K-bit from F.b
        F' ← descendant reached by x' from F
        F^(K).child[x'] ← Aggregate(F')          // set x'-th child
    cache[F] ← F^(K)
return cache[F]
```

is the smallest bit number labeling node F (e.g., $F.b = 0$ at the root of MDD in Fig. 3). To avoid repeatedly visiting the same node, visited nodes are cached; this algorithm assumes that all terminal nodes would have been set to the cache in advance. The time complexity of this algorithm is $O(2^K||F||)$.

Since the maximum height of MDD is shrunk to L/K by bit aggregation, the worst-case search path length is also reduced by a factor of K. This is a great acceleration, but there is a tradeoff between the search time and memory space, as follows.

We set the following assumption about MDD size.

Assumption 2. *The bit aggregation algorithm (Algorithm 1) shrinks the MDD size by a factor of K,*

$$||F^{(K)}|| \approx \frac{||F^{(1)}||}{K}, \tag{9}$$

assuming that MDD nodes are roughly equally distributed over each run of K bits (i.e., bits between $[Ki, K(i+1)-1]$, $i \in [0, L/K-1]$).

Since the memory requirement of MDD $F^{(K)}$ is given by the product of MDD size and memory requirement of each node [30], it is derived with Assumption 2 as follows,

$$\frac{||F^{(1)}||}{K}\big(|b| + 2^K|child|\big), \tag{10}$$

where $|b|$ is the width of bit numbers and $|child|$ is that of child IDs. Given $|b| = |child|$, the memory requirement is minimized at $K = 2$, and increases exponentially for $K > 2$.

The memory requirement might be reduced by introducing heterogeneous MDDs [30], in which non-terminal nodes are allowed to maintain a different number of children and different number of bits. This heterogeneity is also utilized by conventional decision tree methods. However, it prevents efficient MDD traversal introduced in the next subsection, and so we only use MDDs of uniform K in our method.

Algorithm 2: Search

```
Input: F^(K), pkt                                  // pkt is in K-bit array
Output: packet behavior ∈ {I, II, ··· , |P|}
while F^(K) is non-terminal do
    x' = pkt[F^(K).b/K]                            // get K-bit from F^(K).b
    F^(K) = F^(K).child[x']                        // go to next node
return F^(K)                                       // terminal node of behavior
```

5.4 Packet Classification

Algorithm 2 presents the search algorithm that identifies the network-wide behavior of a given packet. It simply follows a path based on the packet. The time complexity is determined just by the maximum height of $F^{(K)}$, that is, $O(L/K)$, because this algorithm includes no operation other than MDD traversal; in contrast, conventional decision tree methods usually involve linear search at a leaf node to select a single rule.

This algorithm is also very efficient in terms of implementation. A packet header is represented as an array of K-bit elements; i.e., i-th K-bit element on the header can be accessed by index i, like $pkt[i]$. The array element itself becomes a child index without fixing numbers of children and bits, which makes this algorithm very efficient in actual implementation. Our algorithm directly handles the raw bit sequence of the packet header by a K-bit array, and so the packet does not need to be pre-processed at all; other implementations might assume that the header fields of interest would be extracted in advance[5].

6 Experiments

This section evaluates our method in terms of memory usage in Section 6.1, construction and update time in Section 6.2, and classification throughput in Section 6.3.

Our method was implemented in C++. The parameter of bit aggregation, K, was chosen from 1, 2, 4, and 8, in order to align K-bit array elements with bytes. The widths of bit numbers and child IDs, $|b|$ and $|child|$, were defined as 32 bits. Our method was compared with HybridCuts [16] as well as a classification method utilizing BDDs [8]. For HybridCuts, parameters "binth" and "spfac" were set to eight and four, which showed the best performance in terms of memory usage and throughput. Search process was added to the original implementation by us. The BDD classification method relies on a set of BDDs of f_i's to determine the network-wide behavior. It was also implemented in C++ by us.

[5] For instance, `http://www.arl.wustl.edu/~hs1/PClassEval.html` and `http://hypercuts.masicek.net/`

Table 1: Statistics of Three Real Networks

	Internet2	Stanford	Purdue
# of switches	9	16	1,646
# of ports used	56	58	2,736
# of rules (FIB)	126,017	757,170	0
# of rules (ACL)	0	1,584	3,605
# of header bits of interest	32	88	104
# of network-wide packet behaviors	86	1,093	10,353

Configuration datasets of the three real networks, Internet2, Stanford backbone network [10], and Purdue campus network [23], were employed in the experiments. The network statistics are shown in Table 1[6]. Configuration rules of FIB (Forwarding Information Base) specify the destination IP field only, while those of ACL (Access Control List) are written as 5-tuple[7]. Some rules specify the VLAN field, but it was ignored in our experiments in order to use a packet trace generator named ClassBench [31], which supports 5-tuple only. This omission, however, has no significant impact on our results, because the MDD size is only 20 % larger at most when the VLAN field is considered.

The experiments were conducted using a single core of Xeon 3.5 GHz with 8 MB cache and 16 GB main memory (DELL PowerEdge Server).

6.1 Memory Usage

Figure 5 shows the MDD size (the number of nodes in an MDD). The size was smaller than the total number of nodes in a set of BDDs, as indicated by (7).

	Internet2	Stanford	Purdue
# of BDD nodes	37,903	547,298	561,635

Figure 5 shows a good fit with (9), which validates Assumption 2; the discrepancy is 9.6 % at maximum (Purdue with $K = 8$).

The amount of memory used by an MDD is shown in Fig. 6, while the size of each MDD node, $|b| + 2^K|child|$, is given in the following table (the memory usage of an MDD is the product of MDD size and node size).

[6]The numbers of network-wide packet behaviors in Table 1 are somewhat different from [8]. We are unable to explain this difference, but in the most complicated Purdue network, our number is greater than that of [8] (3,917) and so this difference does not favor us.

[7]Since Purdue network has no FIB rule, network-wide packet behaviors are defined only at the network edge.

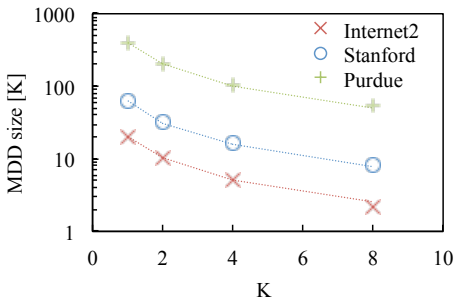

Figure 5: MDD size. The dotted lines indicate (9).

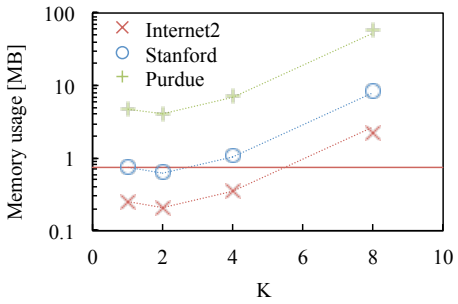

Figure 6: Memory usage of MDD. The dotted lines indicate (10). The horizontal line is memory usage of HybridCuts for Internet2.

K	Node size [B]
1	12
2	20
4	68
8	1028

The memory usage is minimized at $K = 2$ and increases exponentially for larger K, as expected by (10), but it still fits within the CPU cache (8 MB) even for $K = 8$ in Internet2 and Stanford, and for $K = 4$ in Purdue. Although the memory usage is relatively large, 56 MB, at $K = 8$ in Purdue, the CPU cache remained viable for this seven-fold data; this issue is discussed in the throughput evaluation. BDD memory usage is as follows.

	Internet2	Stanford	Purdue
BDD memory usage [MB]	0.454	6.568	6.740

MDD has smaller memory usage than BDD when $K \leq 4$ in Internet2 and Stanford and $K \leq 2$ in Purdue.

HybridCuts successfully constructed a classifier just for Internet2 as described in Section 3, which requires 739 KB of memory. HybridCuts could be comparable to our method if the header space was represented by a small number of hypercubes, but it does not scale with the header space complexity of network-wide packet behavior.

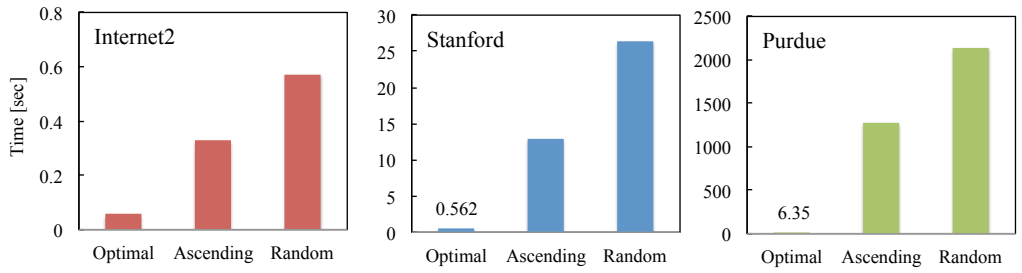

Figure 7: Computation time of MDD unification. Each point is the average of 10 trials.

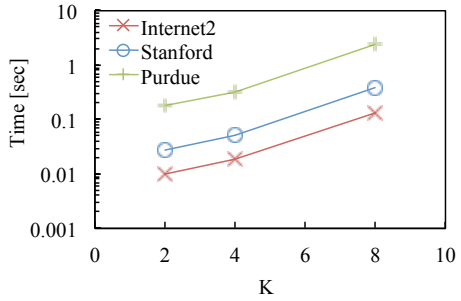

Figure 8: Computation time of bit aggregation. Each point is the average of 10 trials.

6.2 Construction and Update Time

Figure 7 demonstrates the time to unify BDDs of f_i's into a single MDD of $F^{(1)}$ by the operations of (3). We evaluated three calculation orders: the optimal order of Theorem 1, ascending order (a minimum MDD is added to the current unified MDD at every step), and random order. As shown in Fig. 7, the optimal order outperforms the others. Even for the complicated Purdue network, MDDs were unified just in 6.35 sec, which is usually acceptable as an initial construction; note that it can be incrementally updated on demand, as is detailed later.

MDD of $F^{(1)}$ was converted to that of $F^{(K)}$ by the bit aggregation algorithm presented in Algorithm 1. The aggregation time is shown in Fig. 8. As expected, the time grows exponentially with K, but it remained under 2.5 sec which is considered acceptable.

It is worth noting that the calculation time of BDDs is less than 1 sec according to [8], which has to be added to the total construction time of our method.

HybridCuts required 2.17 sec to construct a classifier for Internet2. This is slower than our method with the random calculation order, because HybridCuts processes each hypercube rule individually while our method handles them collectively in a compressed manner.

For MDD updates, we assume that the packet classifier is updated by a new rule associated with a new network-wide behavior. First, the new rule was randomly chosen from the original rules, and then the MDD of classifier was constructed from

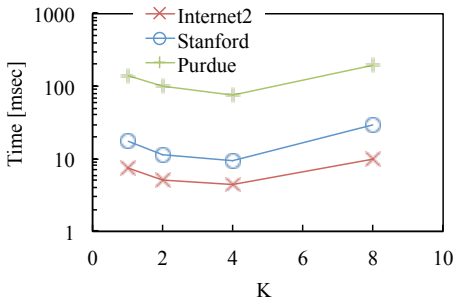

Figure 9: Computation time to update an MDD. Each point is the average of 100 trials.

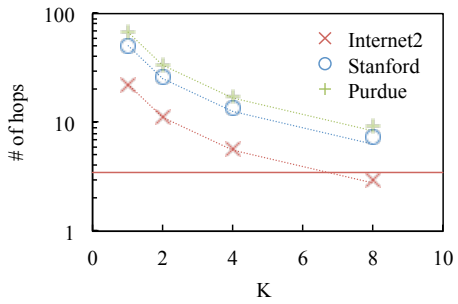

Figure 10: Average hop counts on an MDD to classify a packet. The dotted lines guide $\propto 1/K$. The horizontal line is average hop counts of HybridCuts for Internet2.

the remaining rules. We finally measured the time to perform update operation $\lhd$ over the MDDs of new rule and original classifier. Figure 9 shows the update time. It is less than 10 msec for Internet2, and less than 30 msec for Stanford. Even for Purdue with $K = 8$, the MDD was updated in just 200 msec; these results are entirely acceptable unless time constraints are extraordinarily severe.

6.3 Classification Throughput

Traditionally, classification throughput of packet classification has been evaluated by the number of memory accesses per packet. However, modern processors have complicated architectures with multi-level cache hierarchy, which has a significant impact on throughput, and so classification throughput was measured by actually processing packet headers. In addition to Xeon, classification throughput was also measured using a single core of Core i7 1.7 GHz with 4 MB cache and 8 GB main memory (Apple Macbook Air), in order to examine the impact of hardware.

In the experiments, the classification throughput was measured as follows. First, a million packets with IP and TCP/UDP headers were generated by ClassBench based on the network configurations. The packets were then written into a file in network-byte order. Finally, each packet was read from the file and classified. Note that packets should be generated based on the configurations as done by ClassBench, in order to match all behaviors; randomly generated packets are likely

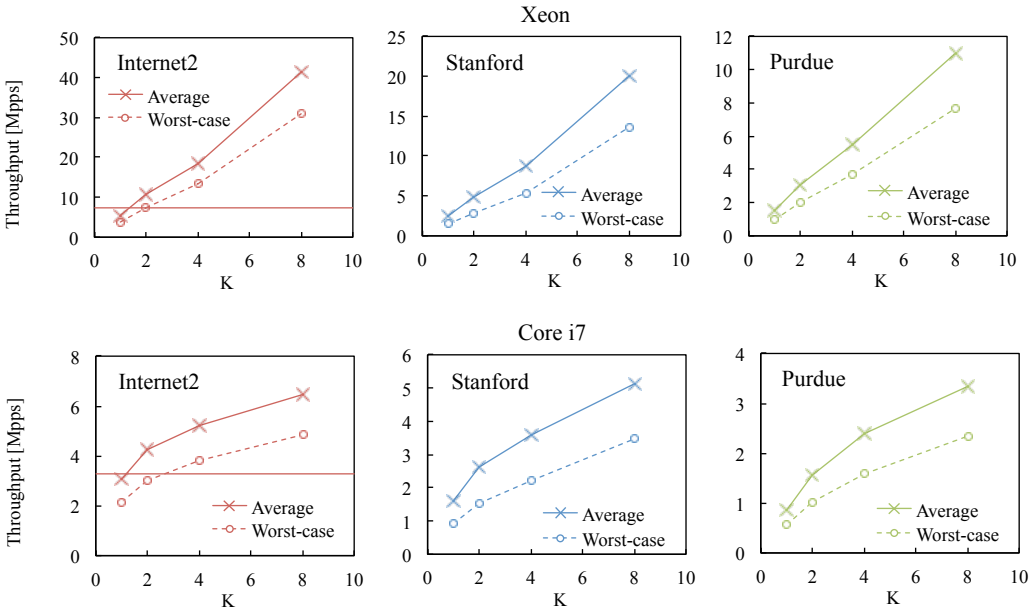

Figure 11: Classification throughput measured on Xeon 3.5 GHz at the top, and on Core i7 1.7 GHz at the bottom. Each cross is the average of 30 trials, and each trial processed a million packets. The worst-case throughput is indicated by dashed lines. The horizontal lines in Internet2 are throughput of HybridCuts.

to match larger subspaces (probably a "default rule"), but never match smaller ones.

Figure 10 shows the average number of hops from the root to a terminal on the MDD of $F'^{(K)}$. Considering that the number of worst-case hops at $K = 1$ is equivalent to the number of bits of interest given in Table 1, the average hop number was roughly 2/3 of the worst-case. The average hop number follows $1/K$ with maximum deviation of 17 % (Stanford with $K = 8$), and it is less than 10 even with $K = 8$ for all networks, which ensures the fast classification of our method. The hop number is much less than the 288 bits of the TCP/IP header, since header fields of no interest are simply skipped on MDDs.

The classification throughput measured on the Xeon is demonstrated at the top of Fig. 11. It exceeded 10 Mpps at $K = 8$ for all networks; at 10 Mpps, even short packets of 125 bytes can fill a 10 Gbps link. The throughput of our method scales linearly against K for all networks, though the MDD is larger than the CPU cache for Purdue with $K = 8$. Figure 11 also shows the worst-case throughput estimated using the worst-case hop number shown in Fig. 10. Even for the worst-case, the throughput is rather high.

Interestingly, the classification throughput shows different behaviors on Core i7, as shown at the bottom of Fig. 11. The throughput is quite high, greater than 3 Mpps at $K = 8$ for all networks, but it does not scale linearly, rather it scales logarithmically.

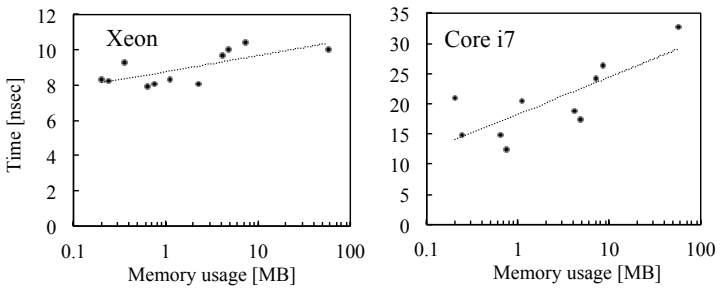

Figure 12: Computation time required for a single hop versus memory usage, measured on Xeon 3.5 GHz at the top and on Core i7 1.7 GHz at the bottom. The dotted line shows the linear regression on $(\log(x), y)$.

This difference in scaling is investigated in depth. Figure 12 shows the computation time required for a single hop versus the memory usage of MDD. The hop time on Xeon stays nearly constant, while that on Core i7 increases logarithmically with MDD size; this is the reason for the scaling difference. We do not dive into the details of CPU architecture and cache hierarchy, which is beyond the scope of this paper, but these results justify our evaluation strategy; the classification throughput should be evaluated based on actual measurements in order to consider the hardware impact.

Throughput of HybridCuts was 7.26 Mpps on Xeon for Internet2, as shown in Fig. 11; it is slower than our method of $K \geq 2$, though the average hop count was quite small, 3.46. This is because each hop took 31.8 nsec, which is nearly four times of our method. HybridCuts has to examine decision criteria at every node, while our simple search algorithm finds the next node just by array access. Moreover, HybridCuts has to find a rule from "binth" number of rules by linear search at a leaf node.

The BDD classification method was slower by several orders of magnitude due to the linear search performed over all behaviors, as follows.

	Internet2	Stanford	Purdue
Throughput by BDDs [Mpps]	0.0849	0.00578	0.00153

7 Discussion from Application Aspects

This section discusses our method in terms of applications. Section 7.1 chooses the best K based on the experiments' results. Section 7.2 discusses the contributions of our method to SDN applications.

7.1 Choice of Best K

Network operators are needed to choose the best K to deploy their applications. Our evaluation in Section 6 revealed a tradeoff between the throughput and con-

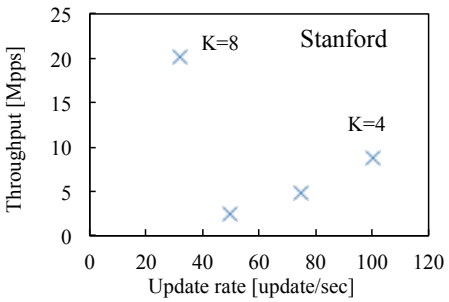

Figure 13: Classification throughput versus update rate. Internet2 and Purdue are omitted since they showed similar results.

struction/update time; large K improves the throughput, but degrades the construction/update time. This tradeoff is shown in Fig. 13 (the construction time is not shown to depict the tradeoff on a two-dimensional plane). The figure tells us that K should be set to 4 or 8, since the points of $K = 4$ and 8 form a *Pareto frontier*, i.e., they cannot be improved without degrading some property. If the network were much simpler, $K = 16$ might be included in the Pareto frontier, but MDDs of $K = 16$ require $128\times$ memory capacity compared to $K = 8$, which would probably make it difficult to fully leverage small CPU caches. In contrast, only $K = 4$ would form a Pareto frontier in more complicated networks.

7.2 Contributions to SDN applications

7.2.1 Network Fabric

Network fabric [3,4] is an idea to improve the economic efficiency and manageability of networks by separating the "intelligence" from the network core. An application program of network fabric constructs a classifier of network-wide packet behaviors based on the network policies. The classifier is then set to edge switches, which tag the header of every incoming packet with some sort of "behavior label". The labeled packets are forwarded through the network core based on the behavior labels. This simple network core can consist of low-cost switches and can be managed just by the labels independently of protocols used in external networks.

To realize the network fabric, the network edge is required to determine the network-wide behavior for every packet at line rate, but that is virtually impossible for conventional classification methods, which failed to construct the classifier or achieved unacceptably low throughput given complex policies like those of the Stanford backbone. Reference [4] estimated that the classification rate of 6.7 Gbps on a single CPU core is enough, and the rate could be enhanced using computer clustering techniques like RouteBricks [32] for a large domain. However, no specific classification method that can deal with network-wide packet behaviors is listed in the paper. Our method is the first one that can realize the network fabric even if the network policy is complex.

7.2.2 Fault Localization

In network fault localization [5], the application software calculates network-wide packet behaviors that could be monitored in the network, in advance. A few probe packets are then selected for each behavior, and the packets are exchanged periodically between switches. If the actually monitored behavior is not identical to the pre-calculated one, the application investigates possible causes as in solving a set cover problem.

Reference [5] assumes that only a small number of predefined packets are employed, since no conventional classification method can classify arbitrary packets. This limitation, which was recognized as a match fault deficiency in [5], implies that faulty behaviors of other packets are overlooked; more importantly, failures occurring on production traffic are not detected. Our classifier allows the application software to examine all packets without ignoring any of them. Our efficient method, of course, does not need to compromise the sampling rate.

7.2.3 Network Verification

Network verification [6–11] is used to check network properties, such as reachability (e.g., packets of a given header subspace sent from a client can get to a specified server), loop-free (no packet in the subspace would traverse any cyclic path), and waypointing (packets from the outside of network should pass through a firewall). The property tests rely on packet classification in order to map the header space to a network-wide packet behavior represented as a directed subgraph.

Assume that in Stanford backbone network, a new rule should be applied immediately to fix a security hole and several properties must be checked with the new rule before deploying it. Subgraphs representing packet behaviors can be updated in 26 msec by [8], while the classifier is updated in 29 msec in our method (Fig. 9); our method is comparably fast. The new behaviors are, then, confirmed with each packet or each header subspace to be tested; assuming that many properties equal to the number of rules in the network, e.g., 100,000, have to be checked, our method only requires 5.0 msec for the lookups while [8] requires 17.3 sec, which can be critical in a severe security incident.

8 Related Work

This section discusses related work; conventional packet classification methods [12–20] are not discussed, since they were thoroughly examined in Section 3.

To fully leverage efficient packet classifiers, fast packet I/O mechanisms should be created to receive packets at line rates. Recent research on system technologies [33] resulted in 14.9 Mpps on a single CPU core, which is roughly equal to our results. Our classification method with the fast packet I/O can provide SDN

applications with the quick identification of network-wide packet behavior on fast links.

OpenFlow[8], one version of SDN, defines the multi-table pipeline; it divides a large classifier into multiple small ones. If the original complicated header space is divided into many parts, the complexity of each part might be greatly reduced. However, the multi-table pipeline focuses on rules specified on a single switch, not network-wide packet behaviors, and it is supposed to handle each dimension with a separate table. Therefore, each machine holds only a few tables (e.g., four tables [34]), and so our problem is not significantly mitigated.

Several papers [35–38], some of which were written in the context of SDN [37,38], discussed algorithms to distribute classification rules among multiple switches without changing their semantics, in order to store the rules in space-limited TCAMs. They are, however, not arranged to identify the network-wide packet behavior.

Multiple-bit striding was utilized to accelerate trie traversal [39, 40], and this technique looks similar with our bit aggregation. Our method is, however, established on MDDs, not simple tries, and so Algorithm 1 is designed to handle path convergence by caching visited nodes and to care skipped nodes of don't care bits.

BDDs have often been used to deal with complicated header spaces along with various network applications such as firewall analysis [41], network state verification [6–8], policy enforcement [42], and traffic analysis [43]. However, work to date did not focus on classification throughput, and so speeds are low. Firewall Decision Diagrams (FDDs) [44, 45] can be used to represent the header space in a compressed manner, but they were designed to be human readable, not to be efficiently manipulated by computers; e.g., the network verifier using FDDs [9] seems to be at least 1,000 times slower than those with BDDs [6, 8]. Prefix DAG [46] employs a data structure similar to MDDs, but it focused on a simple classification problem with a single header field, in which classifiers are readily constructed even if all rules are extracted.

BDDs and MDDs have been intensively studied in the LSI-CAD community. Reference [47] minimized the total size of given BDDs by applying different bit orders to them, while our interest includes the size of MDD. The memory usage of MDD was optimized in [30], but the MDD structure is made complicated and degrading the classification throughput.

9 Conclusions

This paper has developed a packet classification method that efficiently represents the complicated header space defined with network-wide packet behaviors. To fully leverage the global network view provided by SDN, packet classification methods are required to well handle network-wide packet behaviors, not actions taken on

[8] `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`

a single switch. Although packet classification has been studied intensely, conventional classification methods are unable to handle network-wide behaviors at all. Our method is based on compressed decision diagrams and is supported by several new algorithms; it is simple but surprisingly efficient. Numerical experiments on three actual networks showed that our method can identify the network-wide packet behavior at 10 Mpps or more for all three networks. Our work is the only one to solve this hard but important problem in SDN. Moreover, thanks to the introduction of SDN, complex network policies can now be automatically processed by application programs without being manually written in the inefficient 5-tuple format, and so our efficient internal representation will get more opportunities beyond the example applications shown in this paper.

In future work, we will develop SDN applications that utilize our classification method, and evaluate its feasibility with field experiments. Powerful techniques studied in computer science, such as compressed self-indexes [46] and Boolean expression minimization [48], will be applied to our method.

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[2] S. Agarwal, M. Kodialam, and T.V. Lakshman. Traffic engineering in software defined networks. In *IEEE INFOCOM*, pages 2211–2219, 2013.

[3] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A retrospective on evolving sdn. In *ACM HotSDN*, pages 85–90, 2012.

[4] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. Software-defined internet architecture: Decoupling architecture from infrastructure. In *ACM HotNets*, pages 43–48, 2012.

[5] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *ACM CoNEXT*, pages 241–252, 2012.

[6] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. Network configuration in a box: towards end-to-end verification of network reachability and security. In *IEEE ICNP*, pages 123–132, 2009.

[7] R. McGeer. Verification of switching network properties using satisfiability. In *IEEE ICC*, pages 6638–6644, 2012.

[8] Hongkun Yang and S.S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, pages 1–11, 2013.

[9] A.R. Khakpour and A.X. Liu. Quantifying and querying network reachability. In *IEEE ICDCS*, pages 817–826, 2010.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, pages 113–126, 2012.

[11] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, pages 15–28, 2013.

[12] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, pages 213–224, 2003.

[13] Hyesook Lim and Ju Hyoung Mun. High-speed packet classification using binary search on length. In *ACM/IEEE ANCS*, pages 137–144, 2007.

[14] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. Packet classification algorithms: From theory to practice. In *IEEE INFOCOM*, pages 648–656, 2009.

[15] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing packet classification for memory and throughput. *SIGCOMM Comput. Commun. Rev.*, 40(4):207–218, 2010.

[16] Wenjun Li and Xianfeng Li. HybridCuts: A scheme combining decomposition and cutting for packet classification. In *IEEE HOTI*, pages 41–48, 2013.

[17] Hao Che, Z. Wang, Kai Zheng, and Bin Liu. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Transactions on Computers*, 57(7):902–915, 2008.

[18] A.X. Liu, C.R. Meiners, and E. Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in tcams. *IEEE/ACM Transactions on Networking*, 18(2):490–500, 2010.

[19] A. Bremler-Barr and D. Hendler. Space-efficient TCAM-based classification using gray coding. *IEEE Transactions on Computers*, 61(1):18–30, 2012.

[20] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy. Exact worst case TCAM rule expansion. *IEEE Transactions on Computers*, 62(6):1127–1140, 2013.

[21] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[22] A. Srinivasan, T. Ham, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *IEEE ICCAD*, pages 92–95, 1990.

[23] Yu-Wei Eric Sung, Sanjay G. Rao, Geoffrey G. Xie, and David A. Maltz. Towards systematic design of enterprise networks. In *ACM CoNEXT*, pages 22:1–22:12, 2008.

[24] Mark H. Overmars and Frank A. van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21(3):629–656, 1996.

[25] Subhash Suri, Tuomas Sandholm, and Priyank Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[26] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[27] Ryo Yoshinaka, Jun Kawahara, Shuhei Denzumi, Hiroki Arimura, and Shin'ichi Minato. Counterexamples to the long-standing conjecture on the complexity of BDD binary operations. *Information Processing Letters*, 112(16):636–640, 2012.

[28] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms Part 1*, volume 4A. Addison-Wesley, USA, 2011.

[29] David A Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[30] S. Nagayama and T. Sasao. On the optimization of heterogeneous MDDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(11):1645–1659, 2005.

[31] D.E. Taylor and J.S. Turner. ClassBench: a packet classification benchmark. In *IEEE INFOCOM*, volume 3, pages 2068–2079 vol. 3, 2005.

[32] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *ACM SOSP*, pages 15–28, 2009.

[33] Luigi Rizzo. netmap: a novel framework for fast packet I/O. In *USENIX ATC*, pages 101–112, 2012.

[34] Heng Pan, Hongtao Guan, Junjie Liu, Wanfu Ding, Chengyong Lin, and Gaogang Xie. The FlowAdapter: Enable flexible multi-table processing on legacy hardware. In *ACM HotSDN*, pages 85–90, 2013.

[35] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. In *ACM SIGCOMM*, pages 351–362, 2010.

[36] Chad R. Meiners, Alex X. Liu, Eric Torng, and Jignesh Patel. Split: Optimizing space, power, and throughput for TCAM-based classification. In *ACM/IEEE ANCS*, pages 200–210, 2011.

[37] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *ACM CoNEXT*, pages 13–24, 2013.

[38] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *IEEE INFOCOM*, pages 545–549, 2013.

[39] Jahangir Hasan and T. N. Vijaykumar. Dynamic pipelining: Making IP-lookup truly scalable. *SIGCOMM Comput. Commun. Rev.*, 35(4):205–216, 2005.

[40] Yi Wang, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng, Huichen Dai, Xin Tian, Zhonghu Xu, Hao Wu, and Di Yang. Wire speed name lookup: A GPU-based approach. In *USENIX NSDI*, pages 199–212, 2013.

[41] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. Fireman: a toolkit for firewall modeling and analysis. In *IEEE S&P*, pages 199–213, 2006.

[42] Yu-Wei Eric Sung, Carsten Lund, Mark Lyn, Sanjay G. Rao, and Subhabrata Sen. Modeling and understanding end-to-end class of service policies in operational networks. In *ACM SIGCOMM*, pages 219–230, 2009.

[43] Lihua Yuan, Chen-Nee Chuah, and P. Mohapatra. ProgME: Towards programmable network measurement. *IEEE/ACM Transactions on Networking*, 19(1):115–128, 2011.

[44] Mohamed G. Gouda and Alex X. Liu. Structured firewall design. *Computer Networks*, 51(4):1106–1120, 2007.

[45] A.X. Liu and M.G. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1237–1251, 2008.

[46] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. Compressing ip forwarding tables: Towards entropy bounds and beyond. *SIGCOMM Comput. Commun. Rev.*, 43(4):111–122, 2013.

[47] Amit Narayan, Adrian J. Isles, Jawahar Jain, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *IEEE/ACM ICCAD*, pages 388–393, 1997.

[48] Kirill Kogan, Sergey Nikolenko, William Culhane, Patrick Eugster, and Eddie Ruan. Towards efficient implementation of packet classifiers in SDN/OpenFlow. In *ACM HotSDN*, pages 153–154, 2013.