The Journal of Supercomputing (2021) 77:8911–8926
https://doi.org/10.1007/s11227-020-03598-z

KDB: a fast update and high speed packet classifier in SDN

Rashid Hatami[1] · **Hossein Bahramgiri**[1]

Accepted: 23 December 2020 / Published online: 29 January 2021

Abstract
Packet classification is a fundamental function to support several services of software defined networking (SDN). Increasing complexity of the flow tables in SDN leads to challenges for packet classification on update and classification time. In this paper, we propose KDB, a hybrid decision tree classifier, to achieve fast update and high speed packet classification. Experimental results show that KDB is faster in update time compared with SmartSplit and PartitionSort, two state-of-the-art decision tree classifiers, and achieves comparable classification time. Compared with Tuple Search Space (TSS), a classifier used in Open vSwitch, KDB is faster in classification time and achieves comparable update time.

Keywords Packet classification · Software defined networking (SDN) · Decision tree

1 Introduction

Software defined networking (SDN) defines a new paradigm that separates the data plane and control plane. It makes a programmable network using the network applications running in the control plane. To inform the applications of the events that occur in the networks, the central controller interacts with network devices of different architectures through common interfaces. OpenFlow [1] is a standard protocol of SDN that defines the communication between the central controller and network devices. OpenFlow uses flow tables, the data structures that consist of a number of flow entries, and a set of standard feathers to perform instructions of the controller. Flow tables allow the network devices for choosing how to deal with incoming packets. Flows are defined as an equivalence class of packets that contain a subset of packet header fields [2]. The

✉ Hossein Bahramgiri
bahramgiri@mut.ac.ir

Rashid Hatami
rhatami@mut.ac.ir

1 Maleke Ashtar University of Technology, Tehran, Iran

ultimate target of the controller is to specify flows and write into hardware and software flow tables. The flow's behaviors are determined by the controller in the form of rules.

Packet classification is a key function to specify a matching rule on the incoming packet and apply the appropriate action to the packet. It is beyond simple forwarding and provides the requirements of services in SDN [3]. The rule update and classification time are two challenges in the packet classification [4]. The rule updates frequently occur by the controller to perform requirements of network applications such as configuring the forwarding behavior [5]. It is necessary to reduce the update time since it can impact upon the network utilization. The classification time is also important to be minimized to perform a fast process of the services requirements, such as quality of service.

Some proposals focus on reducing update time, such as Tuple Space Search (TSS) [6], or minimizing classification time, such as SmartSplit [7], or simultaneously supporting fast update and classification like PartitionSort [8]. TSS is a fast update method and is used in Open vSwitch [9]. It focuses on minimizing update time while sacrificing classification time. SmartSplit is a high speed classification method which is built upon the state-of-the-art decision trees. It focuses on minimizing classification time while sacrificing memory consumption and update time. SmartSplit gets special attention to keep the tree as balanced as possible to take logarithmic search time. PartitionSort is a state-of-the-art decision tree classifier that develops Multi-dimensional Interval Tree (MITree) to support fast update and high speed classification. PartitionSort partitions a rule set into smaller sortable rule sets and then stores each sortable rule set into MITree. PartitionSort outperforms TSS and SmartSplit in terms of classification speed and update time, respectively [8].

In this paper, we propose a packet classification method called KDB Classifier, which is a hybrid data structure based on KD-tree [10] and B-tree [11]. The motivation of using KD-tree and B-tree is stated as follows. The KD-tree partitions a large search space into a small number of regions. It passes nodes in the tree and returns a region that rule is contained. Each region contains a small number of rules which are stored into a B-tree. The rules can be stored into B-tree in any order and need not to be rebuilt for rule updates. We evaluated our approach using comparisons between other methods: TSS, SmartSplit and PartitionSort. Our results show that KDB achieves fast update and high speed classification. The maximum update time in KDB is 1.33 μs that is comparable to that of TSS and 1.5 times faster than PartitionSort while SmartSplit does not support update. In the worst case, KDB's classification time is 0.75 μs that is comparable to that of PartitionSort, 5.6 times faster than TSS and 3.4 times larger than SmartSplit.

The rest of the paper is organized as follows. Section 2 presents related work. We introduce the proposed algorithm in Sect. 3. In Sect.4, we provide experimental results. Section 5 concludes the paper.

2 Related work

Previous work on packet classification can be divided into three categories: TCAM-based approaches, algorithmic and partitioning methods. TCAMs [12, 13] are not scalable with respect to the rule set size. They can only be utilized in small classifiers. Furthermore, the update time of the TCAM is large because, for example, inserting a new rule in TCAM usually entails rearranging the existing entries. It is also an expensive and power-hungry resource.

Algorithmic methods, such as decision trees and hash-based solutions, can be a viable alternative to overcome the limitations of TCAM. Among algorithmic methods, decision trees are widely studied. These methods partition the search space into regions until each region includes a small number of rules. In decision tree, the root covers the whole searching space and use the packet header for traversing from root to leaf. HiCuts [14] and HyperCuts [15] are two classical decision tree-based methods that work by cutting the total space into several equal-size sub-spaces. These methods suffer from poor uniformity of the sub-space distribution. Indeed, the equal-size cutting inflicts several unneeded sub-spaces that can result in rule duplication. HyperSplit [16] uses the unequal-size splitting to avoid the nonuniformity of the sub-spaces. However, this splitting increases the height of the decision tree and results in a high memory access compared to HiCuts and HyperCuts. SmartSplit [7] initially categorizes the rules into *small* and *large* ranges based on the same fields and then produces a HyperSplit tree for each of them. SmartSplit tries to minimize the classification time by using different algorithms, but it cannot be quickly updated and suffers from inefficient rule update.

Partitioning methods such as Tuple Space Search (TSS) [6] partition the large rule set into a collection of small rule sets for easier management. TSS reduces the scope of exhaustive search by grouping a rule set to smaller rule groups based on the prefix lengths. For each group, one hash table is constructed to insert and delete the rules quickly. TSS supports fast update, but it is slow in classification since a large number of groups must be searched for each packet. In addition, a variety of hash tables for different types of rule sets yields performance variability.

PartitionSort [8] is a hybrid method that is proposed based on the algorithmic and partitioning methods. Similar to partitioning methods, it initially partitions the rule set into smaller subsets and, similar to algorithmic methods, produces a balanced search tree for each subset. PartitionSort is not memory efficient, and its memory consumption increases dramatically with the size of the rule set.

3 The proposed algorithm

In this section, we describe our KDB classifier that focuses on fast update and high speed classification. Our approach can be summarized in the following steps:

– ***Step 1***: *partitioning the search space to achieve fast classification.* In order to reduce classification time, a partitioning technique is used for separating the search space into smaller regions (sub-spaces).
– ***Step 2***: *constructing an efficient data structure to achieve fast update.* After partitioning the search space, we get several regions that include a set of rules. In order to achieve logarithmic time complexity and decrease the difficulties of updates, we consider an efficient data structure for representing the rules of each region.

We next describe the KD-tree and B-tree that will be used in our approach, and following that we introduce KDB.

3.1 KD-tree

In this section, we describe KD-tree [10] as a partitioning technique to separate the searching space into several small regions. A KD-tree is a type of binary tree with the following properties:

– Each node X has k keys that will be called $K_0(X), ..., K_{k-1}(X)$.
– Each internal node contains two pointers, which are either null or reference to their children.
– There is one discriminator associated with each level of the tree that is an integer between 0 to $k-1$. It is utilized to specify the comparison element in each node of the tree.

Let j be the discriminator of node X in KD-tree. If L_{ch} is the left child of X, then $K_j(L_{ch}) < K_j(X)$. If R_{ch} is right child of X, then $K_j(R_{ch}) > K_j(X)$. Each level of the KD-tree includes a same discriminator. The top level of the tree (root node) has discriminator 0, the next level down has discriminator 1, and so the kth level has discriminator $k-1$. In the $k+1$th level of the tree, the discriminator 0 is considered again. This process is carried out recursively. In general, the next discriminator for the level i can be defined as $(i+1)$ *mod* k. Figure 1 shows a 2D-tree for a 2-dimensional search space. The range of regions in the search space is stored as nodes in a 2D-tree. The root node partitions the search space into two subregions based on the discriminator 0. For each subtree of the root, the partitioning is carry out based on the corresponding discriminator recursively.

The search in KD-tree is performed depending on the comparison element in each node. This element is specified by the node's discriminator. The traversal direction of the tree will be forwarded to its left child when the arriving data packet's header is smaller than the comparison element, and to the right child of the tree otherwise. For example, in Fig. 1, assume that a packet $P(f_0,f_1) = (27, 45)$ arrives, where f_0 and f_1 are two fields of the packet's header. In the first step, f_0 is compared with the discriminator "0" of the root node, which is smaller. Thus, the traversal direction is forwarded to the left child of the root node (B node). The next step, f_1 is compared with the discriminator "1", which is smaller again, and the direction is forwarded to

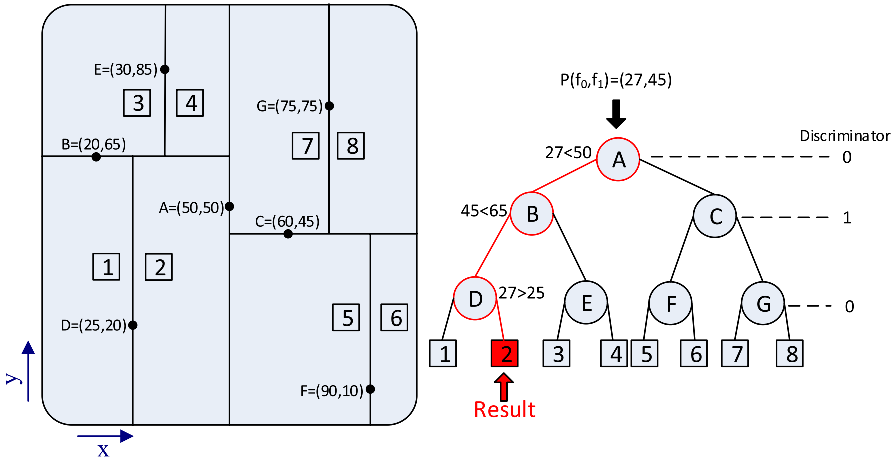

Fig. 1 An example 2-dimensional search space and its 2D-tree

the left node (*D* node). In the node *D*, the comparison between the discriminator "0" and f_0 yields the final result. In this case, the region ID that contains the incoming packet will be returned. The time complexity of the search and insertion algorithm in KD-tree is $O(\log n)$, where n is the total number of the node in KD-tree . Steps to search and insert a node into KD-tree are described in Algorithm 1.

Algorithm 1 KD-tree search and insertion

Input: The root of the KD-tree, a new node X.
Output: Node X is inserted into the KD-tree.

1. **if** (root node is null) **then**
   Set X as the root of KD-tree;
   Set the Left child of X to null;
   Set the right child of X to null;
   Set discriminator of X to 0;
2. **else**
   Move the root node to Q;
3. **end if**
4. **if** (X and Q are equal (i.e. $K_i(X) = K_i(Q)$ for $0 \leq i \leq k-1$) **then**
   X is in the tree;
5. **else**
   X will be left or right child of Q;
   if ($K_i(X) < K_i(Q)$ **then**
     X will be left child of Q;
   else if ($K_i(X) > K_i(Q)$) **then**
     X will be right child of Q;
   end if
   if (Q has not child) **then**
     Go to 8;
   end if
6. **end if**
7. Set child of Q into Q and go to 4.
8. Insert X into KD-tree as a child of Q.
   Set the Left child of X to null.
   Set the right child of X to null.
   Set discriminator of X to $(i + 1 \ mod \ k)$.

3.2 B-tree

In this section, we describe B-tree [11] as a balance search data structure. B-tree has the following properties:

- Each node contains n keys that are stored in nondecreasing order, so that $key\,1 \leq key\,2 \leq ... \leq key\,n$.
- Each internal node includes $n + 1$ pointers to its children.
- The leaf nodes have no children and are at the same depth in the tree.

The details of the operations B-TREE-SEARCH, B-TREE-INSERT, and B-TREE-DELETE are presented in [17]. The time complexity of the search, insertion and deletion algorithms of the B-tree is $O(t \log_t n')$, where n' is the number of keys and t is the minimum degree of the tree.

The search in B-tree is performed like a binary search tree, except that instead of 2-way, a multi-way branching decision at each node is made. The search is based on the range of the keys stored in each node. To insert a new key into B-tree, traverse the tree to reach the leaf node where the new key should be added. If the leaf node is not full, then insert a new key in it and update the order of the keys. If the leaf node is full, then split the node into two new nodes (left and right nodes) based on the

Flow	Field 1	Field 2	Action
F1	011	100	A1
F2	111	001	A2
F3	011	011	A3
F4	110	110	A4
F5	010	111	A5
F6	011	110	A6
F7	001	100	A7
F8	001	110	A8
F9	001	010	A9
F10	101	010	A10

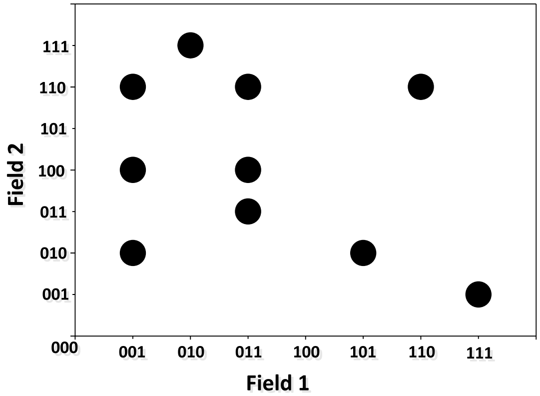

Fig. 2 An example of the switch's flow table and its 2-dimensional space

median key. Afterward, move up the median key to its parent to identify the dividing point. If the parent node is full and it is not the root, then promote the median key of the parent. If the node is root, then create a new root node from the median key to its parent node. The deletion algorithm in B-tree is similar to the insertion, with merging instead of splitting. During the deletion process, it may be needed to rearrange the children of the internal node. Also, a node should not be too small due to deletion.

3.3 KDB classifier

Consider a flow-table contains N flow entries $\{F_i | i = 0, 1, ..., N - 1\}$, each flow is defined on K fields. Using the geometrical view, these flows can be viewed as points in a K-dimensional space. Figure 2 shows an example of the switch's flow table that each flow with two 3-bit fields is located in a 2-dimensional space. The flows are accessible over the search space by the values of their fields. An efficient technique to find flows in a large search space is to partition it into several smaller regions until the number of flows in each region is significantly reduced, and construct an efficient data structure for each region separately. The overall design of the KDB is shown in Fig. 3. When a packet arrives, KD-tree is traversed to return a region, which includes a small number of flows that are stored into a B-tree. Then, a search among these flows leads to the desired matching.

KDB utilizes the KD-tree as a partitioning approach to organize the flows in a k-dimensional space. Figure 4 shows the space partitioning and its corresponding KD-tree for the searching space in Fig. 2. The space partitioning is based on unequal-size sub-spaces that contain as equal number of flows as possible. A region R is a limited range over the search space. We say that a flow F is mapped to R if the ith filed of F, $\forall i \epsilon \{0, ..., k - 1\}$, is found in the R. For example in Fig. 4, the flow $F(3, 6)$ will map to range $R3$ which is limited by two conditions: $Field\,1 < 4$ and $Field\,2 > 5$
.

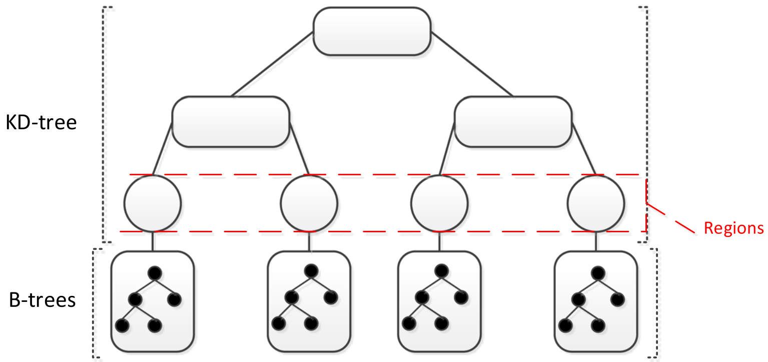

Fig. 3 The overall design of the KDB

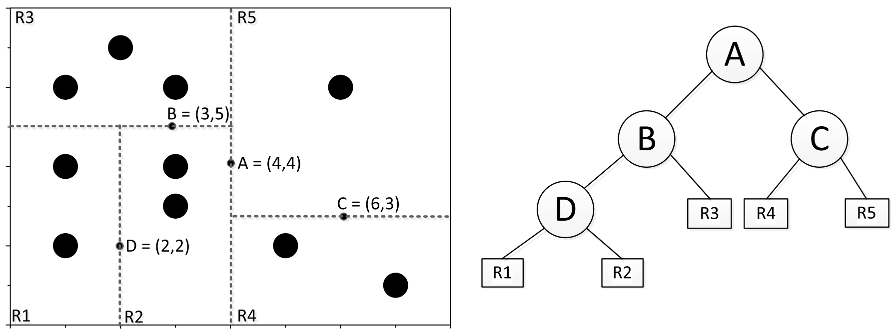

Fig. 4 the space partitioning and its KD-tree in Fig. 2

B-tree is used as an efficient updatable data structure to represent the flows of each region where the flows can be stored in any order and need not to be rebuilt for new updates. To classify packets in KDB, the first step is to specify the region of the incoming packet by traversing over KD-tree, and then refer to the corresponding B-tree that represents the flows of this region. Steps to KDB search are described in Algorithm 2. The update operations (insertion and deletion) are performed over the corresponding B-tree by B-TREE-INSERT/DELETE algorithms. The time complexity in KDB is $O(KD - tree\ time\ complexity + B - tree\ time\ complexity)$.

Algorithm 2 KDB search

Input: The input packet P.
Output: Rule R.

I1. [KD-TREE-SEARCH]
While (Leaf node is not reached) **do**
Compare P with node's key and move down the correct link;
Return corresponding BID(B-TREE's ID);
I2. [B-TREE-SEARCH]
i = 1;
While($i \leq x.n$ and $P > x.rule_i$) **do**
i = i + 1;
If($i \leq x.n$ and $p == x.rule_i$) **then**
Return ($rule_i$);
Else
Return (B-TREE-SEARCH($x.c_i, P$);

$x.n$**:** is the number of keys stored in node x.
$x.rule_i$**:** is the ith rule stored in node x.
$x.c_i$**:** is the ith pointer of the node x.

4 Experimental results

We compare KDB to three classification methods: SmartSplit, TSS and PartitionSort. We evaluate our method based on standard metrics: classification time, update time, memory consumption and construction time. We use ClassBench [18] to mimic the characteristics of the real rule sets because we do not have access to real rule sets. It includes 12 seed files that are divided into three different categories: access control list (ACL), firewall (FW), and IP chain (IPC). Table 1 shows the properties of the different seeds. For experimental evaluation, we generate the lists of 1*K*, 2*K*, 4*K*, 8*K*, 16*K*, 32*K* and 64*K* rules. The classification time is the time required for classifying the packets. We measure the classification time for 1,000,000 packets after constructing the corresponding classifier. To make a fair comparison, we avoid caching in our implementation. The update time is the time required for performing one rule insertion or deletion. We measure this time for 1,000,000 updates for each classifier so that 500,000 insertions are intermixed with 500,000 deletions. Each classifier contains half of the rules. Selecting the rules is random for rule insertion or deletion. We implemented our method in C++. The three methods used in the experiments are hosted on GitHub.[1] We run 10 trials for each metric and report its values. All experiments are run on a machine with Intel Core i7 CPU@4.00GHz, 8MB Cache and 32G DRAM. The operating system is Windows 7.

[1] https://github.com.

Table 1 The properties of the different rule sets

Set	Scale	Specifications
ACL1	733	Security standard format for firewalls and routers (high wildcard)
ACL2	623	
ACL3	2400	
ACL4	3061	
ACL5	4557	
FW1	288	Security rules format for firewalls (medium wildcard)
FW2	68	
FW3	184	
FW4	264	
FW5	160	
IPC1	1702	Decision tree format for software-based systems (low wildcard)
IPC2	192	

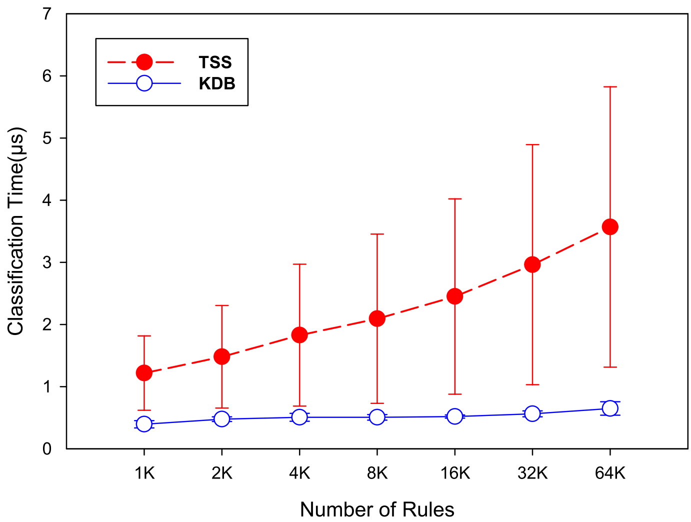

Fig. 5 KDB's classification time versus TSS

4.1 Comparison With TSS

We compare KDB with TSS using the metrics of classification time, update time, construction time, and memory consumption. Figure 5 shows the classification time for KDB and TSS. Experimental results show that KDB is a faster classifier for each size of the rule set compared with TSS. In the worst case, KDB's classification time is 5.6 times faster than TSS. The reason for these results is the large area of TSS since it requires querying a large number of tables. In KDB, a large search space is partitioned into a small number of regions, which include a small number

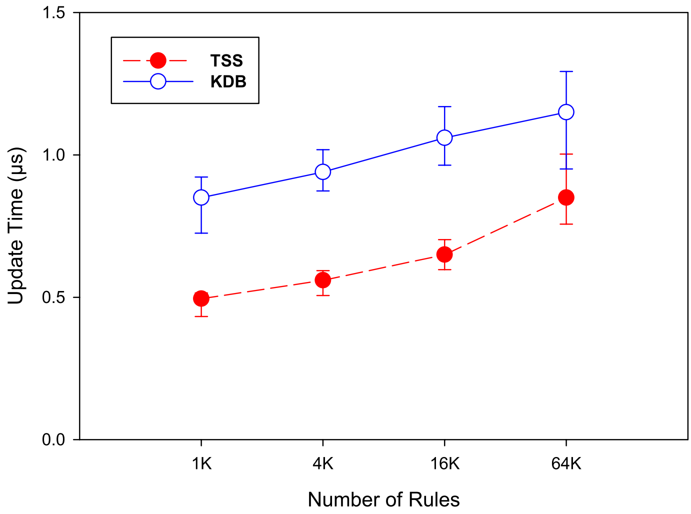

Fig. 6 KDB's update time versus TSS

of rules that are stored into a B-tree. B-tree is a balanced, multi-way search tree that reduces the overall tree height and leads to the reduction in the classification time. Figure 6 shows the update time for KDB and TSS. KDB achieves a fast update with a maximum update time of 1.33 μs. TSS's update is faster than our approach, but it is comparable, and the difference is small. Our experimental results for the largest classifiers show that the construction of KDB is fast with a maximum time of 46 ms. The construction time of TSS is 44 ms that is faster than KDB, but the difference is slightly small. Both KDB and TSS construct all rules of the rule set fast. Figure 7 shows the memory consumption for KDB and TSS. As shown in Fig. 7, for the small classifiers, TSS requires less memory than KDB. As the rule set increases in size, the memory consumption in KDB will be less than TSS. Thus, KDB is more efficient for the larger classifiers compared with TSS.

4.2 Comparison with SmartSplit

We compare KDB with SmartSplit using the metrics of classification time, construction time, and memory consumption. As smartSplit does not support the cases to update rules, we do not consider it in the comparison. Figure 8 shows the classification time for KDB and SmartSplit. Experimental results show that, in the worst case, KDB's classification time is 0.75 μs, that is 3.4 times larger than SmartSplit. As expected, SmartSplit classifies packets faster than KDB. The reason for this result is that SmartSplit uses fewer trees and also it reduces the overall height of the trees by the HyperCuts tree's branching features.

Figure 9 shows the construction time for KDB and SmartSplit. Experimental results show that the construction time of KDB is very fast compared with SmartSplit. SmartSplit's construction time is increased dramatically with the growing

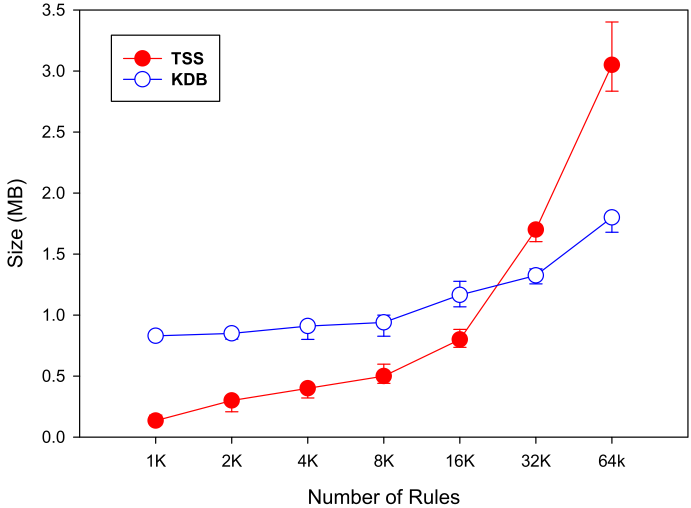

Fig. 7 KDB's memory consumption versus TSS

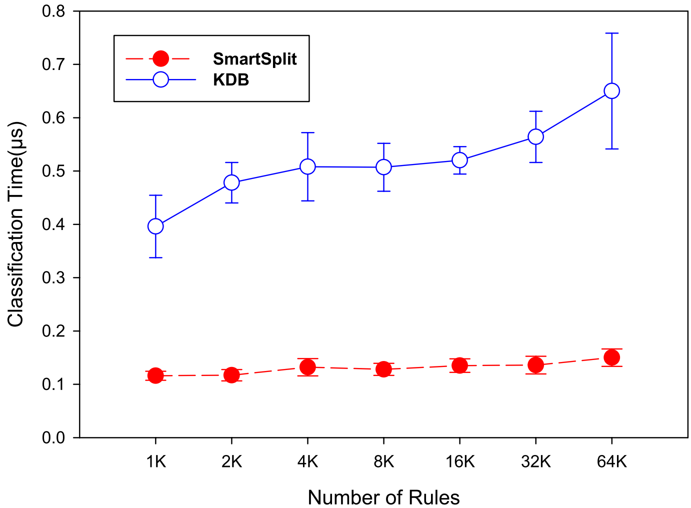

Fig. 8 KDB's classification time versus SmartSplit

of the rules. The construction time for SmartSplit is almost 10 min, while in KDB it is less than a second.

Figure 10 shows the memory consumption for KDB and SmartSplit. The memory consumption in KDB is less than SmartSplit. SmartSplit's memory consumption is very variable. In SmartSplit, depending on the number of rules, type of the tree is different. If the number of rules is small, SmartSplit decides to select single HyperCuts trees. Otherwise, it considers multiple HyperSplit trees.

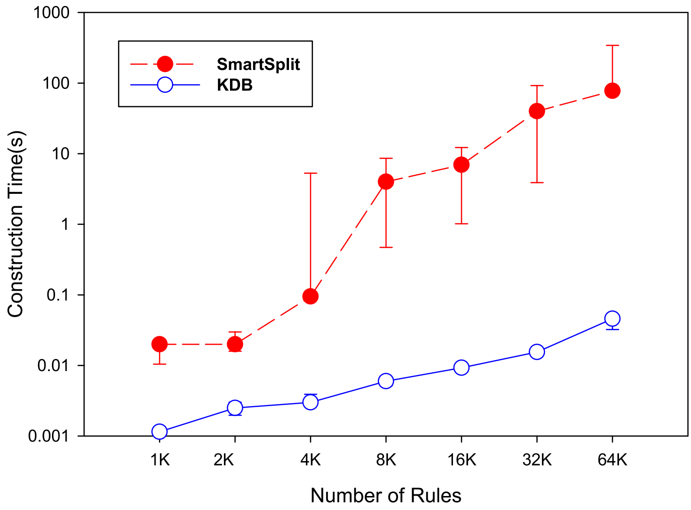

Fig. 9 KDB's construction time versus SmartSplit

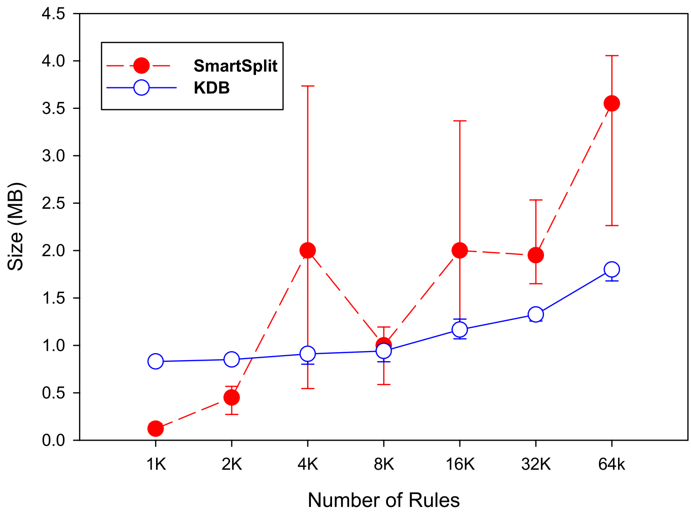

Fig. 10 KDB's memory consumption versus SmartSplit

4.3 Comparison with PartitionSort

We compare KDB with PartitionSort using the metrics of classification time, update time, construction time, and memory consumption. Figure 11 shows the classification time for KDB and PartitionSort. In the worst case, KDB's classification time is about 1.2 times larger than PartitionSort. The reason for these results is the small area of the PartitionSort. PartitionSort requires to query fewer tables compared with KDB.

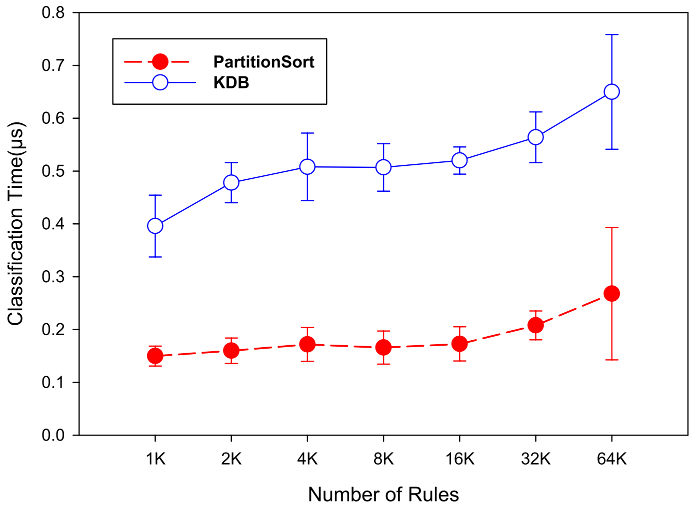

Fig. 11 KDB's classification time versus PartitionSort

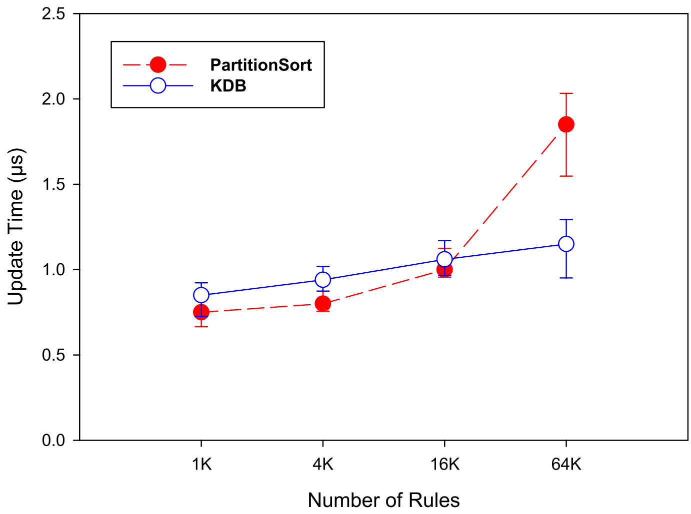

Fig. 12 KDB's update time versus PartitionSort

Figure 12 shows the update time for KDB and PatitionSort. KDB achieves a faster update time compared with PartitionSort. In the worst case, KDB's update time is 1.5 times faster than PartitionSort.

Figure 13 shows the construction time for KDB and PartitionSort. As shown in this figure, KDB's construction is faster than PartitionSort. In the worst case, KDB is built 1.8 times faster than PartitionSort.

In terms of memory consumption, experimental results for the largest classifier show that KDB requires less memory than PartitionSort. Both PartitionSort and TSS

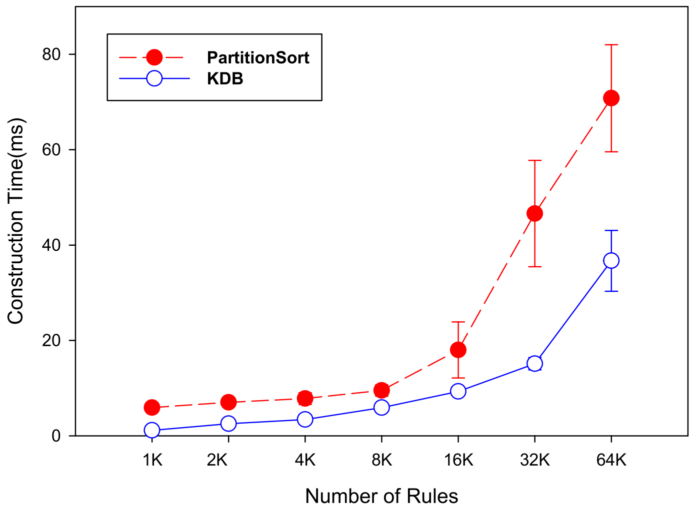

Fig. 13 KDB's construction time versus PartitionSort

Table 2 Comparison with prior art

Approach	Classification (μs)	Update (μs)	Construction (ms)	Memory (MB)
TSS	6.1	1.21	44	3.4
SmartSplit	0.22	500 (s)	500 (s)	4.0
PartitionSort	0.63	2.1	82.9	3.5
KDB	0.75	1.33	46	1.8

are almost the same in memory consumption. Finally, we compare the performance metrics of our approach with the existing methods in Table 2.

5 Conclusion

In this paper, we proposed KDB classifier, a hybrid approach based on KD-tree and B-tree. First, we developed KD-tree as a partitioning technique to separate the large search space into several smaller regions until the scop of search space is significantly reduced. Second, we utilized B-tree as an efficient data structure to decrease the difficulties of updates and support fast classification. Experimental results show that KDB is a fast update and high speed classifier. The maximum update time and classification time in KDB are 1.33 μs and 0.75 μs, respectively.

References

1. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J (2008) Openflow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev 38(2):69–74
2. Wang H, Qian C, Yu Y, Yang H, Lam SS, Wang H, Qian C, Yu Y, Yang H, Lam SS (2017) Practical network-wide packet behavior identification by ap classifier. IEEE/ACM Trans Netw 25(5):2886–2899
3. Li W, Li X, Li H, Xie G (2018) Cutsplit: a decision-tree combining cutting and splitting for scalable packet classification. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, pp 2645–2653
4. Hatami R, Bahramgiri H (2019) High-performance architecture for flow-table lookup in sdn on fpga. J upercomput 75(1):384–399
5. Hatami R, Bahramgiri H (2020) Fast sdn updates using tree-based architecture. Int J Commun Netw Distrib Syst 25(3):333–346
6. Srinivasan V, Suri S, Varghese G (1999) Packet classification using tuple space search. ACM SIGCOMM Comput Commun Rev 29:135–146
7. He P, Xie G, Salamatian K, Mathy L (2014) Meta-algorithms for software-based packet classification. In: 2014 IEEE 22nd International Conference on Network Protocols (ICNP). IEEE, pp 308–319
8. Yingchareonthawornchai S, Daly J, Liu AX, Torng E (2018) A sorted-partitioning approach to fast and scalable dynamic packet classification. IEEE/ACM Trans Netw 26(4):1907–1920
9. Pfaff B, Pettit J, Koponen T, Jackson E, Zhou A, Rajahalme J, Gross J, Wang A, Stringer J, Shelar, et al. P (2015) The design and implementation of open vswitch. In: 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), pp 117–130
10. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
11. Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indices. Acta Inform 1:173–189
12. Lakshminarayanan K, Rangarajan A, Venkatachary S (2005) Algorithms for advanced packet classification with ternary cams. ACM SIGCOMM Comput Commun Rev 35(4):193–204
13. Meiners CR, Liu AX, Torng E (2007) Tcam razor: a systematic approach towards minimizing packet classifiers in tcams. In: 2007 IEEE International Conference on Network Protocols, pp 266–275
14. Gupta P, McKeown N (1999) Packet classification using hierarchical intelligent cuttings. In: Hot Interconnects VII, vol 40
15. Singh S, Baboescu F, Varghese G, Wang J (2003) Packet classification using multidimensional cutting. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. ACM, pp 213–224
16. Qi Y, Xu L, Yang B, Xue Y, Li J (2009) Packet classification algorithms: from theory to practice. In: INFOCOM 2009. IEEE, pp 648–656
17. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms. MIT Press, Cambridge
18. Taylor DE, Turner JS (2007) Classbench: a packet classification benchmark. IEEE/ACM Trans Netw 15(3):499–511