

Article

MultiSplit: An Efficient Algorithm for Packet Classification with Equivalent Priority

Chenshuo Tan ^{1,2} and Zhuo Li ^{1,2,3,*} ¹ School of Microelectronics, Tianjin University, 92. Weijin Rd., Tianjin 300072, China; cstan@tju.edu.cn² Tianjin Key Laboratory of Imaging and Perception of Microelectronics Technology, 92. Weijin Rd., Tianjin 300072, China³ Peng Cheng Laboratory, 2 Xingke 1st St., Shenzhen 518000, China

* Correspondence: zli@tju.edu.cn

Abstract: Packet classification is a core function of network devices for providing advanced services, with the key challenge being to optimize classification speed while maintaining low memory usage. So far, many have proposed software-based packet classification solutions, with most of them adopting a multi-classifier architectures to accommodate the distribution of rule sets. Unfortunately, the need to perform lookups on each classifier during the packet classification stage significantly increases overhead, severely limiting classification speed. To address this shortfall, an efficient packet classification framework based on decision tree algorithms named MultiSplit is proposed. By leveraging the relationships of coverage and priority within the rule set, a new attribute can be abstracted for each rule, termed equivalent priority. Through this preprocessing, MultiSplit significantly reduces redundant lookup overhead while supporting the multi-classifier framework. Additionally, MultiSplit introduces a novel decision tree algorithm that combines multiple splits and intra-level binary search, significantly improving rule separation efficiency. The experimental results show that MultiSplit reduces memory consumption by 49% and decreases memory access by 63%, on average, compared with state-of-the-art packet classification algorithms.

Keywords: packet classification; decision tree; algorithm; OpenFlow



Citation: Tan, C.; Li, Z. MultiSplit: An Efficient Algorithm for Packet Classification with Equivalent Priority. *Electronics* **2024**, *13*, 2967. <https://doi.org/10.3390/electronics13152967>

Academic Editor: Chilukuri K. Mohan

Received: 11 June 2024

Revised: 24 July 2024

Accepted: 25 July 2024

Published: 27 July 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The advancement of computer networks has heightened the demand for sophisticated network equipment. Packet classification is crucial for key functions such as VPNs, firewalls, IDS, and congestion control [1,2]. This process involves a network device like a software switch extracting multi-field information from the incoming packet's header and classifying it based on a specific set of rules. During classification, the packet classification engine performs a lookup to match the packet header with the rule set, determining the relevant policy for enforcement or mapping it to a specific flow [3,4].

To enhance the performance of packet classification in networks, it is crucial to optimize both classification time and memory usage. Faster classification speeds are necessary to meet the throughput demands of high-speed network data planes, while lower memory consumption is essential to address the memory constraints of network devices. Moreover, with the widespread adoption of Software-Defined Networking (SDN), the direct software-based integration of data and control planes has become the norm. This integration imposes specific real-time requirements on packet classification [5,6]. Consequently, packet classification has become a primary focus in modern network research.

Currently, packet classification solutions fall into two broad categories, namely hardware-based and software-based solutions [4]. Hardware solutions typically rely on Ternary Content-Addressable Memories (TCAMs) [7–10]. However, the high costs and significant power consumption associated with TCAMs make hardware solutions less practical for scaling to large-scale classifiers. In comparison, solving packet classification

through the use of software algorithms, such as decision trees [11–22] or hash tuples [23–28], has been widely studied in academia. However, although the performance of the state-of-the-art software algorithm solutions is already outstanding, these solutions still exhibit some limitations. These can be summarized as two main challenges.

First, most algorithms apply a preprocessing step to partition the rule set. Once a rule set is partitioned into subsets, an individual classifier (tuple or decision tree) is constructed for each subset. During packet classification, each packet is queried in all classifiers, even if a matching rule has already been found in one of them, to ensure the identification of all matching rules and select the highest-priority rule as the final match. While many existing algorithms assign priority to classifiers, which allows for pruning during lookup, this only reduces the need for lookups in some classifiers, not all. Therefore, processing packets across multiple classifiers often emerges as the key factor limiting speed improvements in classification.

Second, in decision tree-based algorithms, the depth of the decision tree often directly influences the extent of memory consumption and memory accesses required during classification. However, existing algorithms tend to generate structures with excessively deep trees. Figure 1 illustrates the memory consumption and the depth of the tree structure constructed by HyperSplit [13] (The rule set in the figure is generated from ClassBench [29]. For detailed information such as type and format, please refer to Section 4.1). It is evident that as the rule set expands, the depth of the resulting decision tree grows significantly. Moreover, with a linear increase in tree depth, memory usage almost exponentially increases. Additionally, while cut-based decision tree algorithms decompose the rule space into k subspaces in construction, resulting in a multi-branch decision tree of a more manageable depth, the uniformity of cuts introduces rule replication issues [16,18,25] due to the limitations of equal-sized cutting.

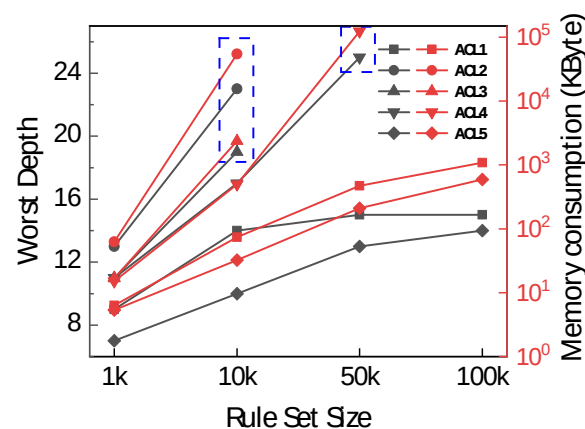


Figure 1. Memory consumption and tree depth of HyperSplit. Note that in some cases, the memory consumption is too large to allow the algorithm to run, resulting in some missing data.

To address the issue outlined above, we first analyze the coverage relationships within the rule set and observe that the overlap among rules directly determines the potential for matching the same packet. Based on this, by assigning a new attribute named *equivalent priority* to each rule to signify its overlap with others, it can efficiently prune redundant queries during the packet classification phase, thereby enhancing classification speed.

In this paper, we propose MultiSplit, a practical packet classification framework based on decision tree algorithms designed to maintain low memory consumption while ensuring high classification speeds. The overall framework of MultiSplit is constructed in three phases. Initially, MultiSplit employs a preprocessing step called equivalent priority grouping, which eliminates uncertainty in matching the same packet by separating rules based on their overlap. Secondly, MultiSplit partitions the rule set based on the prefix length of the rules, segregating rules with longer prefixes from those with shorter prefixes

into different subsets to prevent rule replication. Finally, MultiSplit constructs decision tree classifiers for each rule subset. To address the challenges associated with existing decision tree algorithms—such as excessively deep tree structures, poor scalability, and the potential for memory explosion when dealing with large rule sets—MultiSplit introduces multiple splits in the rule space decomposition, resulting in trees with shallower depths and higher branching factors. Since each rule has already been assigned an equivalent priority in the first step, most packets can terminate the search upon finding the first matching rule during classification, thereby avoiding the need to access all classifiers. Therefore, applying the MultiSplit framework can significantly enhance classification speed.

In summary, the main contributions of this paper are as follows:

- By analyzing the overlaps and coverage relationships in the rule set, a new attribute termed equivalent priority is introduced. Based on this, each rule is assigned an equivalent priority to divide the rule set into several equivalent groups, which enables the omission of numerous redundant lookups during packet classification.
- A novel decision tree algorithm for packet classification called MultiSplit is proposed, which can construct a tree with shallow depth and a high branching factor. By incorporating multiple splits and intra-level binary search, MultiSplit maintains low memory consumption and minimizes memory accesses during queries.
- MultiSplit demonstrates excellent performance in terms of memory consumption, memory access, and classification throughput on various types of rule sets. The experimental results show that MultiSplit reduces memory access by 63% and lowers memory consumption by 49% compared with state-of-the-art algorithms.

2. Background and Related Work

In this section, the background of packet classification is first introduced. Subsequently, for decision tree-based algorithms, the theoretical principles for solving packet classification problems and representative algorithms are presented. Finally, hash-based algorithms and machine learning-based algorithms are discussed.

2.1. Background

The basic process of packet classification involves constructing a classifier based on a set of rules, where packets are matched to apply actions according to their headers as they pass through the classifier. These rules form a list, each consisting of a priority, several matching fields, and a corresponding action. In conventional IPv4 networks, the fields for these rules are usually made up of a standard five-tuple comprising Source/Destination IP Addresses (SIP/DIPs), Source/Destination Ports (SP/DPs), and a transport protocol. In the first generation of OpenFlow switches, the header of a TCP flow is defined by a 10-tuple, including additional fields like VLAN ID that apply to all traffic on a specific VLAN [30]. For ease of discussion in our work, an example rule set with two fields is shown in Table 1.

Table 1. A two-field example classification rule set.

Rule #	Priority	Field X	Field Y	Action
R_1	6	000*	0***	Forward
R_2	5	001*	0***	Forward
R_3	4	00**	****	Enqueue
R_4	3	01**	010*	Forward
R_5	2	01**	011*	Forward
R_6	1	01**	****	Enqueue
R_7	0	****	****	Drop

* is a wildcard, meaning it can represent either 0 or 1.

Notably, a valid rule set must adhere to a fundamental principle, namely that the matching range of a higher-priority rule should not encompass that of a lower-priority rule. The rationale behind this restriction is straightforward. Consider the following example: if rule R_3 in Table 1 is changed from $\langle 00**, **** \rangle$ to $\langle 0***, **** \rangle$, the rule set becomes illegal.

This is because the matching range of rule R_3 will then include rules R_4 , R_5 , and R_6 . Consequently, any packet that matches rules R_4 , R_5 , or R_6 would first be matched by rule R_3 . As a result, rules R_4 , R_5 , and R_6 would never match any packet, rendering them *redundant rules*. The rule sets generated by ClassBench [29] adhere to this principle, and it is also crucial for our subsequent research.

2.2. Decision Tree-Based Solutions

A decision tree-based algorithm conceptualizes the packet classification problem from a geometric perspective [4]. The fields in the rules correspond to dimensions in geometric space, with each rule representing a hypercube in an F -dimensional space, and a packet corresponding to a specific point within this F -dimensional geometry. Classifying a packet is equivalent to identifying the highest-priority hypercube that contains the point representing the packet [31]. Figure 2a presents the geometric view of the rule set shown in Table 1, where each rectangle (two-dimensional cube) represents a rule. Decision tree-based algorithms recursively decompose the rule space into finer-grained subspaces. In this decomposition process, each space is represented by a node in the decision tree. The root node represents the initial entire rule space, while the leaf nodes represent subspaces that meet specific criteria (e.g., containing fewer rules than a threshold) after a certain number of decompositions. Once the decision tree is constructed, packets can be queried against the decision tree to find matching rules within the leaf nodes, which contain a smaller number of rules. The entire process is approximately illustrated in Figure 2b.

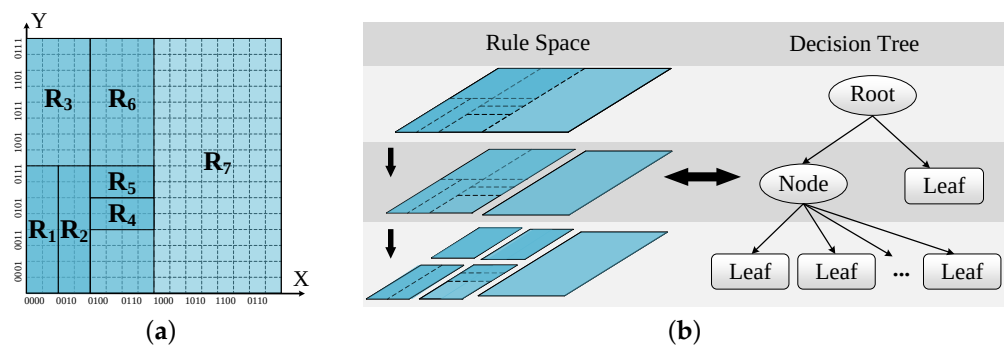


Figure 2. A geometric perspective on packet classification. (a) Geometric model of the two-field rule set shown in Table 1. (b) The basic process of solving the packet classification problem based on the decision tree.

Decision tree algorithms are undergoing active research due to their unique cutting methods and impressive scalability, making them well-suited for multiple-field rule sets. During the iterative decomposition of the rule space, the main node-cutting techniques used in recent work can be categorized as *equal-size cutting*, *equal-dense splitting*, and *bit cutting*.

Equal-sized cutting is a method that uniformly decomposes the search space along a specified dimension, resulting in subspaces that are equal in size. HiCuts [11] is the pioneering algorithm that introduced the concept of building decision trees by decomposing the search space into several equal-sized subspaces to separate rules. However, it only considers cutting along a single field of the rule set, which leads to inefficient cuts and results in a tree with excessive depth. HyperCuts [12] is an improvement of HiCuts, allowing for cuts across multiple fields during each decomposing phase and incorporating optimizations like *node merging*, *rule overlap*, and *region compaction* to construct shorter and fatter trees. Despite these improvements, HyperCuts does not address the rule replication issue in HiCuts, i.e., the inevitable cutting of large-range rules while attempting to separate some small-range ones. Rule replication, which involves cutting a rule internally, is undesirable, as it leads to ineffective rule separation and unnecessarily increases memory usage in nodes. EffiCuts [14] is the first packet classification algorithm that considers the distribution features of a rule set and proposes a pre-partitioning method. EffiCuts suggests

that an effective way to address rule replication is to partition the rule set according to the size of each field before cutting. By doing so, the resulting rule subsets have similar field features, thereby reducing rule replication during the cutting process.

Equal-dense splitting is a method that decomposes the search space along the boundary of a specific rule, resulting in two subspaces that are unequal in size but contain a similar number of rules. HyperSplit [13] introduces the method for non-equal-size cutting named split, aiming to reduce the impact of rule replication by decomposing the search space along rule boundaries. While decision tree structures are meant to be highly scalable, HyperSplit experiences an exponential increase in memory consumption when dealing with large rule sets. ParaSplit [15] introduces the concept of pre-partitioning the rule set before splitting to reduce its complexity. Clustering with range-point conversion in rules can somewhat decrease unnecessary memory usage. However, ParaSplit necessitates tens of thousands of iterations for optimal partitioning, resulting in excessive complexity. CutSplit [18] is a hybrid algorithm that combines the cut and split strategies. It proposes partitioning rules based on the lengths of their field ranges and applies distinct strategies to decompose subsets of rules with different prefix lengths. Additionally, CutSplit combines *pre-cut* and *post-split* to decompose the rule space for the construction of a single decision tree. However, since CutSplit directly employs the HyperSplit algorithm in the post-split phase, its performance can be compromised in certain scenarios.

Bit cutting is a unique method that involves extracting certain bits from rule fields and using this bit string as a directive for cutting. Existing algorithms, such as BitCut [20], ByteCut [21], and MBitTree [22], each utilize different mechanisms to select bits for effective cuts. Rules are decomposed into the same node only if they share the same values at the corresponding bit positions. When the selecting bits contain n bits, cutting generates 2^n child nodes, which may not necessarily be geometrically adjacent within the same rule space.

2.3. Hash-Based Solutions

Hash-based algorithms support fast updates and use linear memory, typically suited for programmable networks where the control plane is separated from the data plane [6,30]. The fundamental concept of hash-based solutions is to group rules with approximately the same effective prefix length into a tuple and then construct a hash table using the field values of the rules as keys. Tuple Space Search (TSS) [23] partitions tuples strictly based on the length of rule fields, with each tuple corresponding to a hash table. Each tuple's match in TSS requires access to all tuples, but rule insertions and deletions need just a single memory access. Therefore, TSS uses linear memory and support for fast updates. TupleMerge [24] enhances TSS by reducing the prefix lengths of rules, which allows it to decrease the number of tuples for faster lookup speeds. However, due to hash collisions, hash-based methods face scalability issues with large rule sets, resulting in lower throughput. DynamicTuple [28] proposes a dynamic programming approach to partition the tuples of the rule set, allowing for the generation of tuples to adapt to the distribution features of the rule set. Additionally, DynamicTuple declares priorities in tuples, entries, and rule lists to achieve effective priority pruning.

2.4. Machine Learning-Based Solutions

Some researchers have proposed using machine learning models to explore the features of rule sets in order to design packet classification algorithms that can adapt to the distribution of the rule sets. NeuroCuts [19] employs deep reinforcement learning to construct decision trees, allowing it to adaptively build more generalized classifiers for various types of rule sets. However, it incurs high training costs and lengthy iteration times. Inspired by *Learned Index* [32], NuevoMatch [33] designs the RQ-RMI model to store disjoint independent rule sets (iSets). The advantage of NuevoMatch lies in its ability to keep storage consumption within a few megabytes, thereby enabling acceleration using the CPU cache. HybridTSS [34] designs a Recursive Tuple Space Search (RTSS) model, which

recursively inserts rules into coarse-grained tuples in a top-down manner. Additionally, it employs Q-learning to determine the dimensions and prefix lengths of partitions, enabling finer-grained partitioning.

3. The Proposed Algorithm

In this section, we propose the MultiSplit packet classification framework, which encompasses a process of grouping based on equivalent priority, pre-partitioning the rule set, and constructing decision tree classifiers using the MultiSplit algorithm. Finally, the packet classification process and related optimizations are introduced.

3.1. Framework

The overall framework of the proposed MultiSplit algorithm is illustrated in Figure 3. This framework consists of the following three main components: “preprocessing”, “construction”, and “classification.” During the execution of the algorithm, these three modules are sequentially applied to complete the entire packet classification process. The function and implementation of each component within the framework are described as follows:

- (a) **Preprocessing:** This part involves executing two processes on the rule set, namely equivalent priority grouping and partitioning. These processes are detailed in Sections 3.2 and 3.3, respectively. First, the rule set is grouped based on the overlap relationships between the rules, ensuring that the rules within each group do not overlap with each other. Each group is then assigned an *equivalent priority*, which is added as a new attribute to each rule. Subsequently, the grouped rule set is partitioned into several rule subsets according to the prefix lengths of the rules.
- (b) **Construction:** This part involves constructing a decision tree-based packet classifier from the preprocessed rule set. For each rule subset, a decision tree is constructed using the MultiSplit decision tree algorithm. The process of constructing the decision tree involves multiple iterations of rule space decomposition, which are detailed in Section 3.4.
- (c) **Classification:** This part involves using the classifier to find the best matching rule for the input packet. Once the classifier is constructed, the packet header is input, and the decision process begins from the root node of the decision tree, traversing level by level until reaching a leaf node. Finally, the matching rule is linearly searched for within the leaf node. During classification, pruning based on equivalent priority attributes significantly accelerates the process. This part is detailed in Section 3.5.

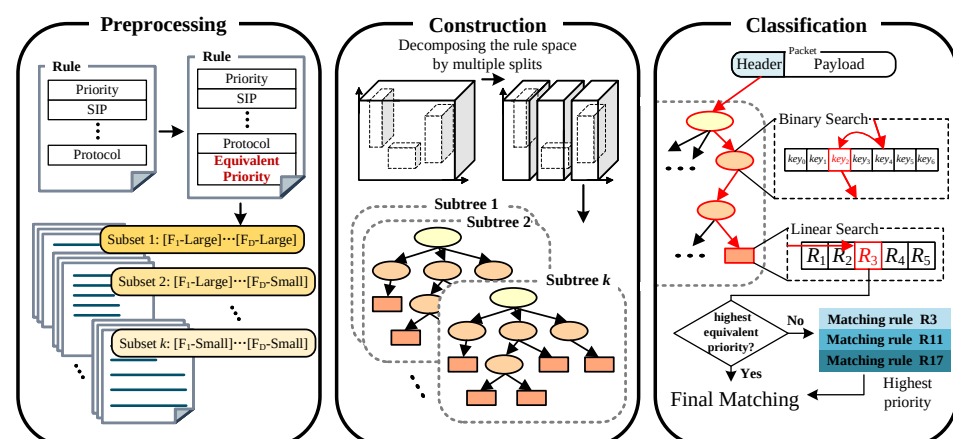


Figure 3. The framework of MultiSplit.

3.2. Equivalent Priority Grouping

During packet classification involving multiple trees/tuples, each packet must identify all applicable matching rules and select the one with the highest priority as the final match.

It has been found that for two rules to match the same packet, they must overlap across all fields. This indicates that there is an inherent abstract relationship between the coverage and the priorities of the rule set. Inspired by [35], a special pattern within the rule set warrants further analysis. Rules with non-overlapping fields can be termed mutually exclusive. Among these, there is no need to debate which rule better matches a packet, as they cannot match the same packet. Therefore, identifying such patterns within the rule set is crucial to minimize the lookup queries across multiple trees/tuples for the highest-priority matching rule.

Taking the five-tuple rule as an example, each field can be represented as a prefix, range, or exact value. This means that a rule can be viewed as a hypercube in five-dimensional space. Similarly, packet headers can be represented by a five-tuple, although the fields in this case contain specific values. Consequently, a packet header can be represented as a point in a five-dimensional hyperspace. If a packet (p) matches both rules R_1 and R_2 , it can be deduced that there is an overlap in each field of rules R_1 and R_2 . Conversely, if at least one field does not overlap between two rules, they are considered non-overlapping, indicating that no packet can match both rules simultaneously. The necessary and sufficient condition for rules R_i and R_j to overlap is shown in (1).

$$R_i \cap R_j \neq \emptyset \iff \bigwedge_{f=1}^D (I_f \langle i, j \rangle \neq \emptyset), \quad (1)$$

where $I_f \langle i, j \rangle$ represents the intersection of rules R_i and R_j in dimension f , and

$$I_f \langle i, j \rangle \neq \emptyset \iff \max(L_f^i, L_f^j) \leq \min(U_f^i, U_f^j),$$

where L_f^i and U_f^i represent the lower and upper bounds of rule R_i in dimension f , respectively.

Based on the above, it can be established that not all rules have the potential to match the same packet. The key factor here is determining the overlap between them. Consequently, to minimize the lookups during packet classification, it is necessary to separate rules that do not overlap with each other from the rule set. This process is referred to as equivalent priority grouping.

Following this, a new attribute named equivalent priority can be abstracted from the grouping results. The grouping process (i.e., the process of calculating equivalent priorities), as shown in Algorithm 1, includes multiple rounds of iteration. Each iteration involves selecting the highest-priority, non-overlapping set of rules from the current rule set, removing it from the rule set, and treating the remaining rules as a new set for the next iteration (lines 1–14). For example, the first iteration involves determining which rules belong to group 1. In group 1, all rules are non-overlapping with each other, and their priority is higher than that of any rules outside this group. At the start of each iteration, the first rule in the set can be directly assigned to the current group (line 2). For each subsequent rule, the algorithm checks its overlap with all preceding rules, and it is assigned to the current group only if it does not overlap with any rules (lines 3–12). After completing the grouping process, the equivalent priority of each rule can be determined based on its group number, meaning the equivalent priority is the complement of the group number (line 19). For instance, the group number for the rule with the highest equivalent priority is 1, the next highest is 2, and so on, with 0 indicating an uncalculated equivalent priority. However, it is unnecessary to compute the equivalent group for all rules, even if the number of such groups in certain rule sets is not large. A predetermined number of equivalent groups (M ; usually 8) is set. After the $M - 1$ iteration, if there are still rules left in the set, they are all assigned to the last equivalent group (lines 15–17).

The results of the equivalent priority grouping for the rule set in Table 1 are illustrated in Figure 4. In the first iteration, rule R_1 is initially placed in group G_1 . Among the remaining rules, R_2 , R_4 , and R_5 do not overlap with any of the preceding rules and are, thus, placed in group G_1 . In the second iteration, considering only the rule set containing R_3 , R_6 , R_7 , rule R_3 is first placed in group G_2 . Subsequently, only rule R_6 does not overlap with any preceding

rules and is also placed in group G_2 . In the third iteration, with only rule R_7 remaining, it is directly placed in group G_3 . Finally, each rule is assigned an equivalent priority based on its group number. Rules $\{R_1, R_2, R_4, R_5\}$ are given an equivalent priority of 2, rules $\{R_3, R_6\}$ are given an equivalent priority of 1, and rule R_7 is given an equivalent priority of 0.

Algorithm 1 Calculating equivalent priorities.

Input: A rule set R contains N rules, the number of groups M

Output: A rule set R' contains N grouped rules

```

1: for  $i = 1$  to  $M - 1$  do
2:   Put the rule  $R_0$  into  $G_i$  ▷  $G_i$ : The  $i$ -th group
3:   for  $j = 1$  to  $n - 1$  do ▷ Current rule set contains  $n$  rules
4:     for  $k = 0$  to  $j - 1$  do
5:       if CHECKOVERLAP( $R_j, R_k$ ) == TRUE then ▷ Call the function on line 22
6:         break
7:       end if
8:       if  $k == j - 1$  then
9:         Put the rule  $R_j$  into  $G_i$ 
10:      end if
11:    end for
12:  end for
13:   $R \leftarrow (R - G_i)$  ▷ Exclude the rules of  $G_i$  from the rule set  $R$  and update  $n$ 
14: end for
15: if  $n \neq 0$  then
16:   Put the remaining rules into  $G_M$ 
17: end if
18: for  $i = 0$  to  $N - 1$  do
19:    $R_i.ep \leftarrow (M - F(R_i).num)$  ▷  $R_i.ep$ : equivalent priority of  $R_i$ ;  $F(R)$ : Find the group to which rule  $R$  belongs
20: end for
21: return  $R'$ 
22: function CHECKOVERLAP( $R_k, R_n$ )
23:   for  $f = \text{SIP to PROTOCOL}$  do ▷ Judge each field
24:     if  $\max(L_f^k, L_f^n) > \min(U_f^k, U_f^n)$  then ▷ According to (1)
25:       return FALSE
26:     end if
27:   end for
28:   return TRUE
29: end function
  
```

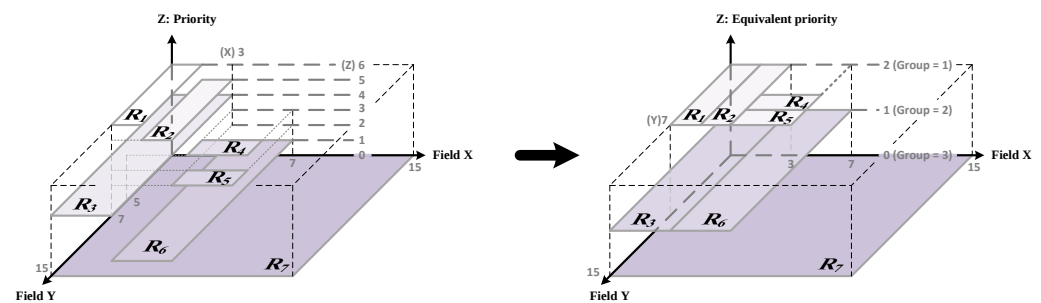


Figure 4. Equivalent priority grouping from a geometric perspective for the example rule set in Table 1. After grouping, the rules located in the same Z plane do not overlap with each other and cannot be matched by the same packet.

Compared with the original priority, the equivalent priority exhibits two new properties. First, different rules can share the same equivalent priority. Second, the total number of equivalent priorities is significantly reduced. The resulting rule proportions under different groups after applying equivalent priority grouping to rule sets generated by

ClassBench [29], represented by the Cumulative Distribution Function (CDF), are shown in Figure 5a. The figure shows that in both ACL and IPC rules, over 60% belong to group 1, with group 1 accounting for as much as 99.4% in the ACL_100k rule set. Furthermore, by applying equivalent priority grouping to the OpenFlow rule sets generated by ClassBench-ng [36], the proportion of rules under different groups is obtained, as shown in Figure 5b. It can be seen that in the OF2 rule sets, over 80% of the rules belong to group 1. Grouping with equivalent priority can significantly reduce redundant lookups during the packet classification stage, which is elaborated on in Section 3.5.

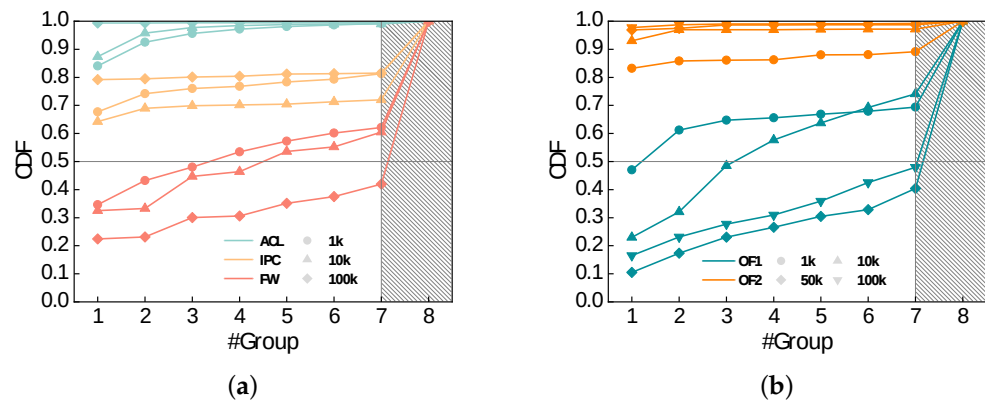


Figure 5. Equivalent priority grouping results of various rule sets. (a) 5-tuple rule sets generated by ClassBench. (b) OpenFlow rule sets generated by ClassBench-ng.

Some existing works also utilize rule overlap, such as SAX-PAC [37]; however, this differs from the equivalent priority grouping proposed in this paper. In terms of implementation purpose, SAX-PAC uses rule overlap to construct the *order-independent* classifier, aiming to optimize the encoding and representation of rules to reduce TCAM space requirements. In contrast, this paper aims to reduce the number of lookups during packet classification to decrease memory access. Regarding implementation principles, SAX-PAC constructs an order-independent by finding the maximum number of rules in the original rule set when each pair of its rules do not intersect. In contrast, this paper groups the rule set based on the original priority order and overlap, with all groups having a priority sequence. In terms of results, the largest order-independent classifier constructed by SAX-PAC accounts for 99.8%, 91.4%, and 96.9% of the ACL1_50k, FW1_50k, and IPC1_50k rule sets, respectively. This significantly differs from the results presented in Figure 5a of this paper.

3.3. Partitioning For Rule Sets

A key issue with constructing classifiers using decision tree algorithms is rule replication [16,18,25]. The primary cause of rule replication is the unavoidable cutting of large-range rules to separate small-range ones. This leads to some large-range rules being cut internally, resulting in these rules being replicated into different child nodes. The impact of rule replication is more detrimental in shallower nodes, as the replicated rules in these nodes are further duplicated in deeper nodes, causing the rule replication factor [18] to increase exponentially.

The distribution of rule sets has specific features that can be utilized to mitigate the impacts of rule replication. For instance, the prefix length of prefix-type rules determines their length in the rule space. By cutting only rules with similar lengths, the likelihood of rules being cut internally can be reduced. Therefore, similar prefix lengths are used to partition the rule set [14,17,18,25] to effectively reduce rule replication in this paper. For N -dimensional rules, a threshold array ($T[N]$) is defined. For each field (F_i), if the field's range length exceeds $T[i]$, it is considered a large field; otherwise, it is a small field. By categorizing rules based on various combinations of large and small fields, the rule set can be partitioned into several subsets, each subset with similar field sizes.

More specifically, since the SIP/DIP fields offer more distinctiveness in rule differentiation and excessive partitioning into subsets can actually increase the complexity of classification, partitioning is only performed on the SIP/DIP fields. After partitioning the rule set, a maximum of 2^2 subsets of rules are created. Decision tree classifiers are constructed for each rule subset to minimize rule replication as much as possible. Figure 6 illustrates the prefix distribution and partition scheme of the IPC_10k rule set. When $T[2] = \{20, 20\}$, the IPC_10k rule set (total of 914 rules) is partitioned into 4 subsets. The first subset contains 37 rules satisfying SIP prefix length ≤ 20 and DIP prefix length ≤ 20 . The second subset consists of 153 rules satisfying SIP prefix length ≤ 20 and DIP prefix length > 20 . Similarly, the third subset contains 132 rules, and the fourth subset contains 592 rules. Additionally, each subset is identified by the highest equivalent priority of the rules within it to facilitate subsequent pruning.

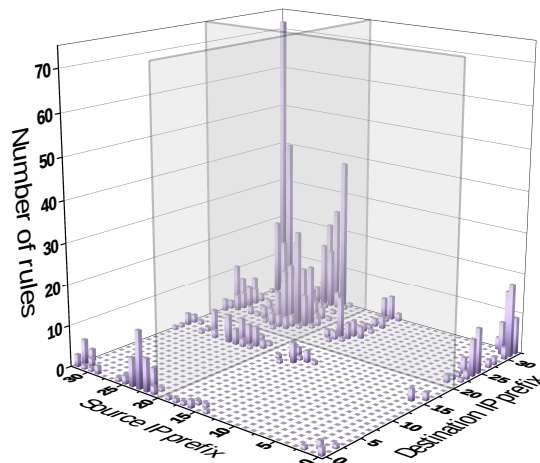


Figure 6. Prefix distribution and partition scheme of IPC_10k rule set.

3.4. Decision Tree Construction

As described in Section 2, cut and split are two mainstream methods for decomposing the rule space in decision trees, with the split method opting to separate rules at their boundaries. This method somewhat alleviates the rule replication issue encountered by the cut method. However, its binary mechanism leads to low efficiency in rule separation. More specifically, the hierarchical index of the decision tree constructed using the cut method is shown in (2), where $p(f)$ denotes the value of a packet in the f -th field and $Node(f).Pt$ represents the comparison key of a node in dimension f . The hierarchical index of the decision tree constructed using the split method is shown in (3), where $Node(f).Min$ indicates the lower bound of the rule space of a node in dimension f , and $Node(f).\Delta$ denotes the cutting step size within the node. Both of these hierarchical indexing operations require only limited computations or comparisons, with a complexity of $O(1)$. In comparison, the former algorithm enables indexing across multiple child nodes in a single hierarchical indexing stage, while the latter is confined to indexing within just two child nodes. This suggests that the conventional split method's low efficiency in separating rules can lead to unnecessarily complex tree structures, such as those with excessive depth, ultimately resulting in significant memory consumption and even memory explosion.

$$index = \begin{cases} 0 & , p(f) < Node(f).Pt \\ 1 & , p(f) \geq Node(f).Pt \end{cases} \quad (2)$$

$$index = \left\lceil \frac{p(f) - Node(f).Min}{Node(f).\Delta} \right\rceil \quad (3)$$

Based on this, MultiSplit introduces multiple splits in each iteration process, creating multiple child nodes in rule space decomposition. The whole process of decision tree construction by MultiSplit is shown in Algorithm 2. For a given rule space, MultiSplit

executes a set of fixed decomposition strategies to construct a single level of the decision tree and then iteratively applies the same process to the resulting subspaces for further decomposition. The decomposition strategy is as follows. The N rules (R_i) in this rule space (S) are projected onto dimension f and mapped to N segments (Seg_R). By extracting the end points (up to $2N$) of these segments and merging overlapping end points, each pair of adjacent end points forms a new segment, resulting in a total of M segments Seg_P (line 4–6). Then, the following three-step heuristic is executed.

Algorithm 2 MultiSplit decomposition strategy

Input: A rule space $S = \{R_1, \dots, R_N\}$ contains N rules, leaf node rule number threshold $binth$, decomposition factor spf and step factor β

Output: np decomposed rule spaces C_i or current rule space S

```

1: if  $N > binth$  then
2:    $MinNum \leftarrow 2N + 1$ 
3:   for  $f = SIP$  to  $PROTOCOL$  do
4:      $Seg_R \leftarrow PROJECT(R, f)$   $\triangleright Seg_R$ : one-dimensional segments of rules
5:      $P_R \leftarrow EXTRACT(Seg_R)$   $\triangleright P_R$ : the endpoints of the segments(may be overlap)
6:      $(Seg_P, P_P) \leftarrow MERGE(P_R)$  and  $SORT(P_P)$   $\triangleright$  Get  $M_s$  segments and  $M_P$  points
7:      $AveNum_f \leftarrow \frac{1}{M_s} \sum_{i=0}^{M_s-1} Seg_P^i.rn$   $\triangleright Seg_P^i.rn$ : number of rules covering  $Seg_P^i$ 
8:     if  $AveNum_f < MinNum$  then
9:        $f_{sel} \leftarrow f$   $\triangleright$  Select  $f$  as the splitting dimension
10:       $MinNum \leftarrow AveNum_f$ 
11:    end if
12:  end for
13:   $np \leftarrow 2$   $\triangleright np$ : the number of decomposed subspaces
14:  while true do
15:    for  $i = 0$  to  $np - 1$  do
16:      for  $j = 0$  to  $M_s$  do
17:        if  $\sum_{k=0}^j Seg_P^k.rn > \frac{i+1}{np} \sum_{k=0}^{M_s} Seg_P^k.rn$  then
18:           $P_{split}^i \leftarrow P_P^j$   $\triangleright$  Choose  $P_P^j$  as the position of  $i$ -th split
19:          break
20:        end if
21:      end for
22:       $C_i \leftarrow SPLIT(S, P_{split}^i)$   $\triangleright C_i$ : the  $i$ -th subspace
23:    end for
24:    if  $\frac{1}{N} (\sum_{i=0}^{np-1} C_i.rn + np) < spf$  then
25:       $np \leftarrow \beta \times np$ 
26:    else
27:      break
28:    end if
29:  end while
30:  return  $C = \{C_0, \dots, C_{np-1}\}$   $\triangleright C$ : all decomposed subspaces
31: else
32:  Terminate decomposition and designate current rule space  $S$  as a leaf node
33:  return  $S$ 
34: end if

```

1. Choosing the dimensions: For each dimension, calculate the average number of rules covering each segment (Seg_P) as $AveNum_f$. The dimension with the minimum $AveNum_f$ is then selected as the dimension for executing the subsequent split (line 7–11).

2. Picking the number of splits: Initialize the number of nodes after splitting np as two (line 13); then, calculate the position of each split point as described in step (3). Assuming a hyperplane decomposition of the rule space at these points, np subspaces are produced. For each subspace (C_i), calculate the number of rules contained in each rule space ($C_i.rn$). Then, employ the heuristic discriminant inequality (4) to determine if the current np can be used as the number of decompositions for the rule space. Update np incrementally to $\beta \times np$ and repeat the strategy as long as the condition is not met. Note that since MultiSplit does not use equal-size cutting, it is not necessary to ensure that np is a power of 2 (lines 13–29).

$$\frac{1}{N} \left(\sum_{i=0}^{np-1} C_i.rn + np \right) < spfac \quad (4)$$

3. Calculate the position of splits: Whenever a new np is determined, it is necessary to calculate the split positions for decomposition of the current rule space. Once np is chosen as the number of decompositions for the current node's rule space, the resulting subspaces are treated as child nodes, and this configuration is used as input for the next round of decomposition. On the other hand, to ensure that the hyperplanes used for splitting fall on the boundaries of the rules, it cannot simply decompose the rule space at equal intervals along the selected dimension. Therefore, the positions of the splits are chosen at the end points of Seg_p to ensure that each split effectively separates the rules. For the i -th split, the smallest Seg_p boundary that meets the condition of inequality (5) is selected as the position of the hyperplane for the split (lines 15–23).

$$\sum_{k=0}^j Seg_p^{k.rn} > \frac{i}{np} \sum_{k=0}^{M_s} Seg_p^{k.rn} \quad (5)$$

where $Seg_p^{k.rn}$ represents the number of rules covered by the k -th segment.

The above strategy can ensure that each subspace covers a roughly equal number of rules while minimizing the impact of rule replication. Constructing the decision tree with MultiSplit is an iterative process, so the algorithm needs to explicitly define the termination condition for stopping the decomposition of the rule space. In the case that the number of rules in the node is not greater than a threshold called *binth*, the splitting process is stopped.

3.5. Packet Classification

Each rule subset after partitioning generates a decision tree constructed by the building algorithm. A schematic of the data structure for MultiSplit tree nodes and the lookup process in a single node are shown in Figure 7. Looking up in a decision tree starts at the root node and indexes downward level by level. For non-leaf nodes, the packet header and the range segments stored in the node are used to find the index of the next level's child node. Since most non-leaf nodes contain numerous range segments, MultiSplit uses a binary search to index the matching child node. For leaf nodes, a linear search of the node's rule space is conducted to find the matching rule. Among all the matching rules, the one with the highest priority is selected as the final match. Algorithm 3 shows the detailed process of lookup using the MultiSplit decision tree. To avoid unnecessary lookups, the following two optimizations are implemented in MultiSplit.

1. Introduce a tree priority for each tree, setting the highest equivalent priority of the rules within it. During the classifying process, if a matching rule is already found and its equivalent priority is higher than the priority of the next decision tree to be traversed, that tree is skipped.
2. During the classifying process, if a matching rule is found with the highest equivalent priority (i.e., group 1), it is immediately selected as the final matching rule, ending the lookup. Since over half of the rules in the rule set have the highest equivalent priority, this optimization significantly speeds up the packet classification process.

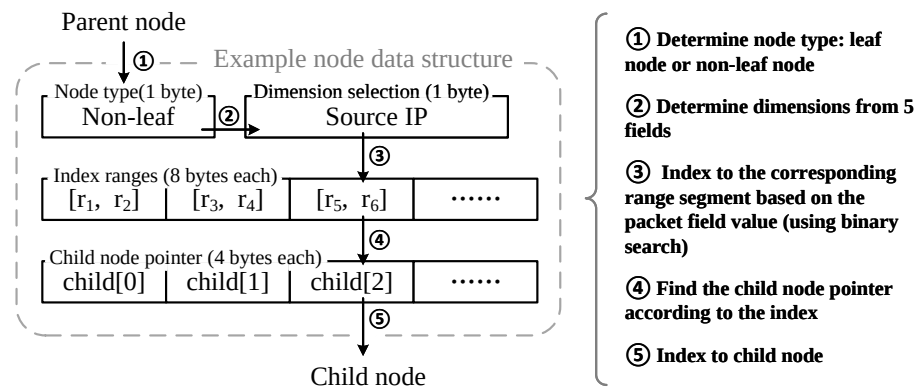


Figure 7. Data structure of tree node and search process.

Algorithm 3 MultiSplit Decision Tree Lookup Algorithm

Input: A packet header $P = \{v_1, \dots, v_D\}$, the root nodes of K decision trees $\{B_1, \dots, B_K\}$

Output: Final matching rule R_{out}

```

1: for  $j = 1$  to  $K$  do
2:   if Equivalent priority of matching rule  $R_{match.ep} \geq$  priority of current decision tree  $B_j.ep$  then
3:     continue
4:   end if
5:    $N \leftarrow B_j$  ▷ Take root node  $B_j$  as the current processing node
6:   while  $N$  is a non-leaf node do
7:      $f \leftarrow$  the specified dimension in node  $N$ 
8:      $i \leftarrow \text{INDEXCHILD}(Seg_P, v_f)$  ▷ Calculate the child node index
9:      $N \leftarrow C_i$  ▷ Take child node  $C_i$  as the current processing node
10:    if  $N$  is an empty node then
11:      break ▷ Skip current tree
12:    end if
13:  end while
14:  if  $N$  is not a null node then
15:     $R_{match} \leftarrow$  linearly search for matching rules in node  $N$ 
16:  end if
17:  if the matching rule has the highest equivalence priority  $R_{match.ep}$  then
18:    return  $R_{out} \leftarrow R_{match}$ 
19:  end if
20: end for
21: return  $R_{out} \leftarrow R_{match} \{\max(R_{match.ep})\}$ 
22: function INDEXCHILD( $Seg_P, v_f$ )
23:    $s \leftarrow 0$ 
24:    $e \leftarrow (M - 1)$ 
25:   while  $s \leq e$  do
26:      $m \leftarrow \frac{s+e}{2}$ 
27:     if  $v_f \in Seg_P^m$  then
28:       break
29:     else if  $v_f > Seg_P^m.U$  then
30:        $s \leftarrow m + 1$ 
31:     else
32:        $e \leftarrow m - 1$ 
33:     end if
34:   end while
35:   return  $m$ 
36: end function

```

The second optimization can significantly accelerate the classification speed during packet classification without affecting the correctness of the classification results. This can be easily demonstrated as follows: suppose a packet (P) matches rule R_1 , and R_1 has the highest equivalent priority. In this case, R_1 must be the final matching rule. If there exists another rule (R_2) that also matches packet P and has a higher priority than R_1 , it implies that R_1 and R_2 overlap. Consequently, R_1 and R_2 would not be grouped together during the equivalent priority grouping, and the group containing R_2 would have a higher equivalent priority than the group containing R_1 . Therefore, R_1 cannot have the highest equivalent priority, which contradicts the given condition, that is, R_2 does not exist. In fact, our experimental results confirm that the classification results are entirely correct after the rule set is grouped by equivalent priority.

4. Evaluation

In this section, the performance of MultiSplit is evaluated on a PC platform and compared with that of HyperSplit [13], CutSplit [18], MBitTree [22], and TupleMerge [24] in terms of memory access, throughput, and memory consumption. These represent conventional split-based algorithms, the most advanced decision tree algorithms, bit cutting algorithms, and tuple-based hashing algorithms, respectively. The types and features of the various comparison algorithms evaluated in the experiments are summarized in Table 2. Meanwhile, the structural details of the decision tree and the preprocessing time of MultiSplit are also evaluated.

Table 2. Summary of the comparison algorithms.

Algorithm	Type	Partitioning Method of Rule Set	Pruning Method During Classification
HyperSplit [13]	Decision tree based on split	No partitioning	No pruning
CutSplit [18]	Decision tree based on FiCut and split	Partitioning by prefix lengths	Pruning by subtree priority
MBitTree [22]	Decision tree based on bit cutting	Clustering by prefix lengths	Pruning by subtree priority
TupleMerge [24]	Hash-based tuple	Partitioning by coarse-to-fine prefix	Pruning by tuple priority
MultiSplit	Decision tree based on split	Partitioning by prefix lengths	Pruning by equivalent priority

4.1. Experimental Setup

MultiSplit and all comparison algorithms are implemented in C++, using gcc 9.4.0 with the C++11 standard [38], in the simulation experiments. All algorithms run initialization, preprocessing (including partitioning, construction, etc.), and classification modules in single-thread mode. The parameter settings for each algorithm follow the default settings reported in the original papers. The parameter settings for MultiSplit are as follows: the number of equivalent priority groups is $M = 8$, the threshold for the number of rules in a leaf node is $binth = 8$, the decomposition factor is $spfac = 1.5$, and the stepping factor is $\beta = 1.5$. These settings are chosen based on the optimal rule separation efficiency and minimal rule replication. The performance is evaluated on a PC with an Intel(R) Core(TM) i5-10505 CPU @ 3.20 GHz and 16 GB RAM. The operating system is Ubuntu 20.04.

The rule sets used in the experimental evaluation are synthetic and generated by ClassBench [29]. ClassBench contains 12 seed files for the following three types of rule sets: five Access Control Lists (ACL) seeds, five Firewall (FW) seeds, and two IP Chain (IPC) seeds. In the comparative evaluations in Sections 4.2 and 4.3, we selected one representative seed from each type and generated rule sets of four different sizes (1k, 10k, 50k, and 100k) for each. This resulted in a total of 12 rule sets; for example, ACL_50k represents a rule set generated from an ACL seed with a size of approximately 50,000. In the individual evaluations in Sections 4.4 and 4.5, we used all the seed files to generate rule sets of the

same size. This also resulted in a total of 12 rule sets; for example, FW3_10k represents a rule set generated from the FW3 seed with a size of approximately 10,000.

We also used the *trace_generator* of ClassBench to create trace sets containing simulated traffic. Each trace set corresponds to a rule set and contains a number of packets that is ten times the size of the corresponding rule set. For example, IPC_10k_trace is a test trace set paired with the IPC_10k rule set and contains approximately 100,000 packets.

Note that missing data in the experiments indicates that HyperSplit could not run normally under the specified rule set due to its inability to handle large-scale rule sets. In some cases, the algorithm ran for an extended period before the process was terminated.

4.2. Memory Access and Throughput

This section evaluates the memory access and classification throughput of MultiSplit and other algorithms. Both metrics are evaluated during the packet classification process. The memory access tests the average number of accesses to the classifier data structure needed to classify one packet. This number is counted using a global counter during the algorithm's execution. Note that in all algorithms, accessing a tree node, tuple, or rule counts as one memory access. The classification throughput measures the average number of packets the algorithm can classify per second (in MPPS, million packets per second). This metric is obtained by timing the classification process and calculating the throughput.

Both memory access and throughput are indicators of packet classification time efficiency, with limited memory access being a major challenge when deploying controller software to software switches in network data planes [39]. On the other hand, throughput provides a more direct representation of overall classification efficiency. Figure 8 illustrates the performance of different algorithms in terms of memory access. Evidently, MultiSplit outperforms other algorithms in terms of memory access. Compared with HyperSplit, MultiSplit achieves only 29% of its average memory accesses. Furthermore, MultiSplit achieves just 42% and 37% of the average memory accesses compared with CutSplit and MBitTree, respectively.

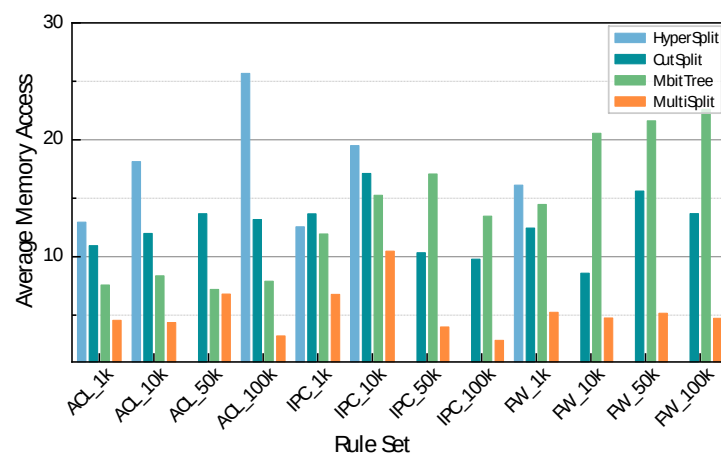


Figure 8. Evaluation results of the average number of memory accesses.

In terms of throughput, as shown in Figure 9, MultiSplit occupies a leading position compared with existing algorithms. Preprocessing using equivalent priority grouping significantly reduces the number of tree lookups during the packet classification stage. On rule sets larger than 10k, MultiSplit achieves a 113% performance improvement over HyperSplit. Compared with other algorithms, MultiSplit consistently holds a leading edge.

MultiSplit demonstrates notable query performance due to the equivalent priority grouping preprocessing applied to the rule set. During the classification process, the optimization method based on equivalent priority significantly enhances classification speed. Additionally, the decision tree constructed by MultiSplit is shallow, with a large branching

factor. Consequently, most packets can reach the leaf nodes by traversing few nodes from the root, enabling rapid querying.

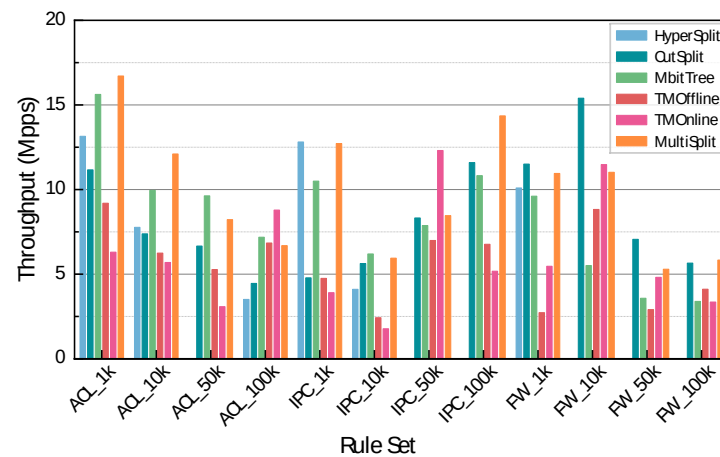


Figure 9. Evaluation results of throughput.

The data reveal that there is not a complete correlation between memory access and throughput in MultiSplit. The reason is that MultiSplit utilizes binary search to index matching child nodes in the decision tree's hierarchical index, which incurs slightly more computational overhead.

4.3. Memory Consumption

In this section, we evaluate the memory consumption of MultiSplit and other algorithms. This metric is measured by the size of the classifier data structure after the algorithm completes classifier construction. Figure 10 illustrates the memory consumption of data structures across various packet classification algorithms. Notably, MultiSplit completely overcomes the memory explosion encountered in scenarios with large rule sets. On the IPC_10k and FW_10k rule sets, the memory usage of MultiSplit is only 21% and 0.8% of that of HyperSplit, respectively. This is attributed to the partitioning the rule set by prefix length and the employment of multiple splits in tree construction, resulting in shorter and wider trees. Compared to other advanced algorithms, the memory consumption of MultiSplit is also at an acceptable level.

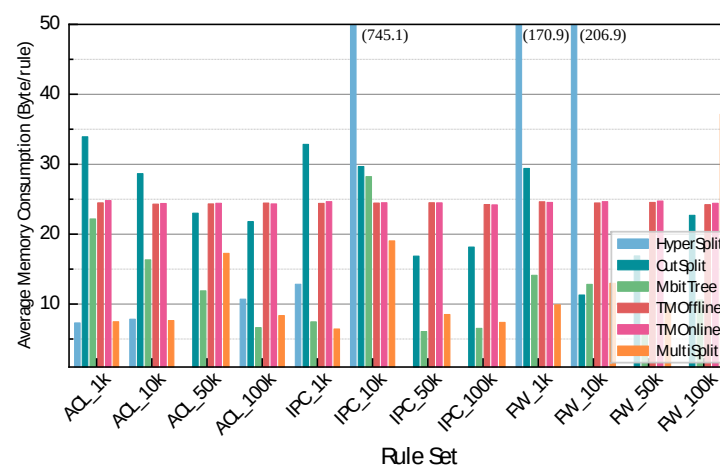


Figure 10. Evaluation results of average memory consumption.

4.4. Tree Depth and Average Branching Factor

To demonstrate the effectiveness of multiple splits, some specific details of the decision trees generated by MultiSplit are measured across different rule sets, including the average

and maximum tree depth and branching factor. As shown in Table 3, it is evident that the tree structure constructed by MultiSplit has an average depth of only 2.6 levels, with an average branch factor of 641. The resulting tree structures display distinctively short and wide features, further achieving fewer memory accesses and enhanced storage efficiency. Additionally, by utilizing intra-level binary search, MultiSplit can accomplish the task of multi-level indexing based on the conventional split-based algorithm with just a single level, thus reducing memory access overhead. This allows MultiSplit to overcome the structural and performance limitations often encountered in conventional algorithms.

Table 3. Tree depth and average branching factor of MultiSplit.

Rule Set	Depth		Branch Factor	
	Max	Average	Max	Average
ACL1_10k	5.33	2.67	849.67	19.67
ACL2_10k	6.75	3.75	1364.00	841.25
ACL3_10k	6.75	2.75	195.75	5.75
ACL4_10k	7.25	3.25	119.75	38.25
ACL5_10k	4.50	2.00	89.00	6.00
FW1_10k	4.25	2.25	1384.25	1382.50
FW2_10k	3.00	1.00	957.50	942.25
FW3_10k	4.50	2.50	936.00	935.00
FW4_10k	6.00	3.25	852.00	851.00
FW5_10k	5.00	2.75	803.75	803.25
IPC1_10k	7.25	2.75	360.50	16.50
IPC2_10k	1.00	1.00	1854.33	1854.33
MEAN	5.12	2.49	813.85	641.31

4.5. Preprocessing Time

The preprocessing time of MultiSplit is then evaluated, including the time for equivalent priority grouping and classifier construction, as shown in Figure 11. For grouping time, MultiSplit can complete the group processing of a 50k rule set in less than 2 s on average, and even for a 100k rule set, it finishes grouping within 8 s. Regarding construction time, MultiSplit can build classifiers for rule sets smaller than 10k in under 1 s. For rule sets larger than 50k, it may take several seconds to tens of seconds to complete construction. As it generally meets the packet classification needs of static rule sets, it is acceptable.

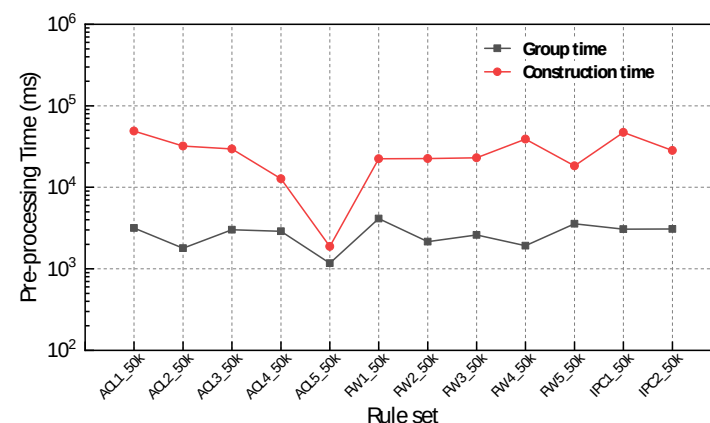


Figure 11. Preprocessing time of MultiSplit in 50k rule sets.

4.6. Discussion

This section provides a comparative evaluation of MultiSplit with existing algorithms in terms of memory access, classification throughput, and memory consumption. Additionally, we conduct an independent evaluation of MultiSplit regarding tree structure details and preprocessing time. The experimental results indicate that the overall classification performance of MultiSplit surpasses that of current algorithms. Although MultiSplit, like

CutSplit and MBitTree, belongs to the “multi-tree” classifier architecture, it employs equivalent priority preprocessing for rule sets, significantly reducing the number of decision tree accesses. This immediate confirmation method for the best matching rule is applicable to most existing packet classifiers, including tuple architectures such as TupleMerge. On the other hand, MultiSplit uses multiple splitting to decompose the rule space, resulting in decision trees with shallow depth and large branching factors. Compared to HyperSplit, where packet decisions are limited to two child nodes at each level of the decision tree, MultiSplit enables decisions among numerous child nodes at a single level. Consequently, MultiSplit overcomes the rule explosion issue in large-scale rule sets through the inclusion of a controllable decision tree structure. Overall, the comprehensive evaluation demonstrates that MultiSplit achieves the anticipated performance improvements.

5. Conclusions and Future Work

5.1. Conclusions

Continuously optimizing classification speed and memory usage represents a significant challenge in addressing packet classification problems through software algorithms. Furthermore, redundant queries in existing multi-classifier architectures are a critical bottleneck limiting enhancements in packet classification speed. To address this shortfall, this paper proposes an efficient packet classification framework called MultiSplit based on decision tree algorithms. By leveraging the relationships of coverage and priority within the rule set, a new attribute can be abstracted for each rule, termed equivalent priority. Through this preprocessing, MultiSplit significantly reduces redundant lookup overhead while supporting the multi-classifier framework. Additionally, MultiSplit introduces a novel decision tree algorithm that combines multiple splits and intra-level binary search, significantly improving rule separation efficiency. The performance of MultiSplit was evaluated, and the experimental results show that its performance in terms of memory access and memory consumption is significantly improved compared with state-of-the-art packet classification algorithms.

As a preliminary part of the network data plane, packet classifiers are generally deployed in software switches, relying on general-purpose processors to map packets to network flows. According to the MultiSplit architecture, their implementation mainly involves the following two steps: first, preprocessing and construction of the classifier data structure for the built-in rule set (or input rule set from the control plane) and, second, classification of the input packets in the switch through the classifier. This deployment method allows MultiSplit to meet the advanced service needs of most enterprise or data center networks.

5.2. Future Work

Despite the outstanding performance of MultiSplit, there are still some shortcomings. Therefore, we plan to undertake the following work in future research. Increasingly large network scenarios rely on network architectures such as Software-Defined Networking (SDN) for management. This may require that rule-set entries be updated in real-time according to control plane policies and that the packet classifier be adjusted in real time according to changes in the rule set. Hence, future research should focus on packet classification algorithms that support rule-set updates. Secondly, to avoid performance limitations and dependency on the CPU core environments in actual deployment, future research should explore packet classification algorithms based on hardware–software co-design. For instance, programmable hardware devices such as FPGAs can be used, leveraging their high-speed parallel processing and low-latency advantages.

Author Contributions: Conceptualization, C.T. and Z.L.; methodology, C.T. and Z.L.; software, C.T.; validation, C.T.; formal analysis, C.T.; investigation, C.T. and Z.L.; resources, C.T. and Z.L.; data curation, C.T. and Z.L.; writing—original draft preparation, C.T.; writing—review and editing, C.T. and Z.L.; visualization, C.T.; supervision, Z.L.; project administration, Z.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data used in the article were generated through open-source platforms.

Acknowledgments: The authors would like to express their gratitude to Xiaolin Gong from the School of Microelectronics, Tianjin University; Wang Zhang and Wanli Zhao from the China Automotive Technology and Research Center; and Heping Shi from the School of Automobile and Transportation, Tianjin University of Technology and Education for their invaluable contributions to project management in this research. The authors would like to take this opportunity to thank the editor, editorial team, and all anonymous reviewers for their effort and time in reviewing the manuscript, providing valuable comments and suggestions, and helping in improving the quality of the manuscript.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

VPN	Virtual Private Network
IDS	Intrusion-detection system
SDN	Software-Defined Networking
TCAM	Ternary Content-Addressable Memory
SIP/DIP	Source/Destination IP address
TCP	Transmission Control Protocol
VLAN	Virtual Local Area Network
TSS	Tuple Space Search
TM	TupleMerge
RQ-RMI	Range Query Recursive Model Index
CDF	Cumulative Distribution Function
ACL	Access Control List
IPC	IP Chains
FW	Firewall
OF	OpenFlow
PC	Personal Computer
FPGA	Field Programmable Gate Array

References

1. Lim, H.; Lee, N.; Jin, G.; Lee, J.; Choi, Y.; Yim, C. Boundary cutting for packet classification. *IEEE/ACM Trans. Netw.* **2013**, *22*, 443–456. [\[CrossRef\]](#)
2. Coscia, A.; Dentamaro, V.; Galantucci, S.; Maci, A.; Pirlo, G. An innovative two-stage algorithm to optimize Firewall rule ordering. *Comput. Secur.* **2023**, *134*, 103423. [\[CrossRef\]](#)
3. Srinivasan, V.; Varghese, G.; Suri, S.; Waldvogel, M. Fast and scalable layer four switching. In Proceedings of the ACM SIGCOMM'98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Vancouver, BC, Canada, 31 August–4 September 1998; pp. 191–202.
4. Taylor, D.E. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv. (CSUR)* **2005**, *37*, 238–275. [\[CrossRef\]](#)
5. Yang, L.; Ng, B.; Seah, W.K.; Groves, L.; Singh, D. A survey on network forwarding in Software-Defined Networking. *J. Netw. Comput. Appl.* **2021**, *176*, 102947. [\[CrossRef\]](#)
6. Kuźniar, M.; Perešini, P.; Kostić, D. What you need to know about SDN flow tables. In Proceedings of the Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, 19–20 March 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 347–359.
7. Rottenstreich, O.; Cohen, R.; Raz, D.; Keslassy, I. Exact worst case TCAM rule expansion. *IEEE Trans. Comput.* **2012**, *62*, 1127–1140. [\[CrossRef\]](#)
8. Rottenstreich, O.; Keslassy, I.; Hassidim, A.; Kaplan, H.; Porat, E. Optimal in/out TCAM encodings of ranges. *IEEE/ACM Trans. Netw.* **2015**, *24*, 555–568. [\[CrossRef\]](#)
9. He, P.; Zhang, W.; Guan, H.; Salamatian, K.; Xie, G. Partial order theory for fast TCAM updates. *IEEE/ACM Trans. Netw.* **2017**, *26*, 217–230. [\[CrossRef\]](#)
10. Srinivasavarma, V.S.; Vidhyut, S. A TCAM-based caching architecture framework for packet classification. *ACM Trans. Embed. Comput. Syst. (TECS)* **2020**, *20*, 2. [\[CrossRef\]](#)
11. Gupta, P.; McKeown, N. Packet classification using hierarchical intelligent cuttings. In Proceedings of the Hot Interconnects VII, Stanford, CA, USA, 18–20 August 1999; Volume 40.

12. Singh, S.; Baboescu, F.; Varghese, G.; Wang, J. Packet classification using multidimensional cutting. In Proceedings of the 2003 Conference On Applications, Technologies, Architectures, and Protocols for Computer Communications, Karlsruhe, Germany, 25–29 August 2003; pp. 213–224.
13. Qi, Y.; Xu, L.; Yang, B.; Xue, Y.; Li, J. Packet classification algorithms: From theory to practice. In Proceedings of the IEEE INFOCOM 2009, Rio de Janeiro, Brazil, 19–25 April 2009; pp. 648–656.
14. Vamanan, B.; Voskuilen, G.; Vijaykumar, T. EffiCuts: Optimizing packet classification for memory and throughput. *ACM SIGCOMM Comput. Commun. Rev.* **2010**, *40*, 207–218. [\[CrossRef\]](#)
15. Fong, J.; Wang, X.; Qi, Y.; Li, J.; Jiang, W. ParaSplit: A scalable architecture on FPGA for terabit packet classification. In Proceedings of the 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, Santa Clara, CA, USA, 22–24 August 2012; pp. 1–8.
16. Li, W.; Li, X. HybridCuts: A scheme combining decomposition and cutting for packet classification. In Proceedings of the 2013 IEEE 21st Annual Symposium on High-Performance Interconnects, San Jose, CA, USA, 21–23 August 2013; pp. 41–48.
17. He, P.; Xie, G.; Salamatian, K.; Mathy, L. Meta-algorithms for software-based packet classification. In Proceedings of the 2014 IEEE 22nd International Conference on Network Protocols, Raleigh, NC, USA, 21–24 October 2014; pp. 308–319.
18. Li, W.; Li, X.; Li, H.; Xie, G. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications, Honolulu, HI, USA, 16–19 April 2018; pp. 2645–2653.
19. Liang, E.; Zhu, H.; Jin, X.; Stoica, I. Neural packet classification. In Proceedings of the SIGCOMM '19: Proceedings of the ACM Special Interest Group on Data Communication, Beijing, China, 19–23 August 2019; pp. 256–269.
20. Liu, Z.; Sun, S.; Zhu, H.; Gao, J.; Li, J. BitCuts: A fast packet classification algorithm using bit-level cutting. *Comput. Commun.* **2017**, *109*, 38–52. [\[CrossRef\]](#)
21. Daly, J.; Torng, E. Bytecuts: Fast packet classification by interior bit extraction. In Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications, Honolulu, HI, USA, 16–19 April 2018; pp. 2654–2662.
22. Tan, J.; Lv, G.; Qiao, G. MBitTree: A fast and scalable packet classification for software switches. In Proceedings of the 2021 IEEE Symposium on High-Performance Interconnects (HOTI), Santa Clara, CA, USA, 22–24 August 2021; pp. 60–67.
23. Srinivasan, V.; Suri, S.; Varghese, G. Packet classification using tuple space search. In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Cambridge, MA, USA, 30 August–3 September 1999; pp. 135–146.
24. Daly, J.; Bruschi, V.; Linguaglossa, L.; Pontarelli, S.; Rossi, D.; Tollet, J.; Torng, E.; Yourtchenko, A. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM Trans. Netw.* **2019**, *27*, 1417–1431. [\[CrossRef\]](#)
25. Li, W.; Yang, T.; Rottenstreich, O.; Li, X.; Xie, G.; Li, H.; Vamanan, B.; Li, D.; Lin, H. Tuple space assisted packet classification with high performance on both search and update. *IEEE J. Sel. Areas Commun.* **2020**, *38*, 1555–1569. [\[CrossRef\]](#)
26. Zhang, C.; Xie, G. MultilayerTuple: A General, Scalable and High-performance Packet Classification Algorithm for Software Defined Network System. In Proceedings of the 2021 IFIP Networking Conference (IFIP Networking), Espoo and Helsinki, Finland, 21–24 June 2021; pp. 1–9.
27. Zhong, J.; Wei, Z.; Zhao, S.; Chen, S. TupleTree: A High-Performance Packet Classification Algorithm Supporting Fast Rule-Set Updates. *IEEE/ACM Trans. Netw.* **2023**, *31*, 2027–2041. [\[CrossRef\]](#)
28. Zhang, C.; Xie, G.; Wang, X. DynamicTuple: The dynamic adaptive tuple for high-performance packet classification. *Comput. Netw.* **2022**, *202*, 108630. [\[CrossRef\]](#)
29. Taylor, D.E.; Turner, J.S. Classbench: A packet classification benchmark. *IEEE/ACM Trans. Netw.* **2007**, *15*, 499–511. [\[CrossRef\]](#)
30. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [\[CrossRef\]](#)
31. Toth, C.D.; O'Rourke, J.; Goodman, J.E. *Handbook of Discrete and Computational Geometry*; CRC Press: Boca Raton, FL, USA, 2017.
32. Kraska, T.; Beutel, A.; Chi, E.H.; Dean, J.; Polyzotis, N. The case for learned index structures. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 489–504.
33. Rashelbach, A.; Rottenstreich, O.; Silberstein, M. A computational approach to packet classification. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, Virtual, 10–14 August 2020; pp. 542–556.
34. Liu, Y.; Xin, Y.; Li, W.; Song, H.; Rottenstreich, O.; Xie, G.; Li, W.; Wang, Y. HybridTSS: A Recursive Scheme Combining Coarse- and Fine- Grained Tuples for Packet Classification. In Proceedings of the 6th Asia-Pacific Workshop on Networking, APNet '22, Fuzhou, China, 1–2 July 2022; pp. 43–49. [\[CrossRef\]](#)
35. Jia, C.; Li, C.; Li, Y.; Hu, X.; Li, J. An Observation of Packet Classification: Most Rules are at the Top. In Proceedings of the IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), New York, NY, USA, 2–5 May 2022; pp. 1–6.
36. Matoušek, J.; Lučanský, A.; Janeček, D.; Sabo, J.; Kořenek, J.; Antichi, G. ClassBench-ng: Benchmarking Packet Classification Algorithms in the OpenFlow Era. *IEEE/ACM Trans. Netw.* **2022**, *30*, 1912–1925. [\[CrossRef\]](#)
37. Kogan, K.; Nikolenko, S.; Rottenstreich, O.; Culhane, W.; Eugster, P. SAX-PAC (Scalable And eXpressive PACket Classification). *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 15–26. [\[CrossRef\]](#)

-
38. ISO/IEC. ISO/IEC 14882:2011: Information Technology—Programming Languages—C++. 2011. Available online: <https://isocpp.org/std/the-standard> (accessed on 10 June 2024).
 39. Tang, L.; Huang, Q.; Lee, P.P. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Trans. Netw.* **2020**, *28*, 2350–2363. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.