applied sciences

MDPI

Article

ITOC: An Improved Trie-Based Algorithm for Online Packet Classification

Yifei Li [1,2], Jinlin Wang [1,2], Xiao Chen [1,2] and Jinghong Wu [1,2,*]

[1] National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, No. 21, North Fourth Ring Road, Haidian District, Beijing 100190, China; liyf@dsp.ac.cn (Y.L.); wangjl@dsp.ac.cn (J.W.); xxchen@dsp.ac.cn (X.C.)

[2] School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, No. 19(A), Yuquan Road, Shijingshan District, Beijing 100049, China

* Correspondence: wujh@dsp.ac.cn

Abstract: With the development of SDN, packet classifiers nowadays need to be provided with low update latency besides fast lookup performance because switches need to respond to update control messages from controllers in time to guarantee real-time service in SDN implementations. Classification in this scenario is called online packet classification. In this paper, we put forward an improved trie-based algorithm for online packet classification (ITOC), in which we provide a trie selection strategy to avoid occasional high update latency in the update process of online trie-based algorithms. Experiments are conducted to validate the effectiveness of our optimization and compare the performance of ITOC with the offline methods, DPDK ACL. Experimental results demonstrate that ITOC has the same level of lookup speed with DPDK ACL and greatly decreased the update latency as well. The update latency of ITOC is only 6.85% of DPDK ACL library in the best case.

Keywords: SDN; SDN switch; packet c; update latency

Citation: Li, Y.; Wang, J.; Chen, X.; Wu, J. ITOC: An Improved Trie-Based Algorithm for Online Packet Classification. *Appl. Sci.* **2021**, *11*, 8693. https://doi.org/10.3390/app11188693

Academic Editor: Juan Francisco De Paz Santana

Received: 5 July 2021
Accepted: 15 September 2021
Published: 17 September 2021

1. Introduction

1.1. Motivation and Problem Statement

As one of the most important parts of network implementation, packet classification plays an important role when we provide network services, such as forwarding, routing, firewall, and some other complex services. Packet classification problems attracted a lot of interest and have been widely studied for the past two decades [1].

However, new demands emerged in the packet classification of software-defined networking (SDN) [2]. SDN decouples the control plane and data plane, brings programmability, and flexibility to the network. SDN implementation heavily relies on SDN switches. SDN switches use the same lookup tables with match-action rules as traditional ones. However, their work scenes are quite different.

In traditional switches, the lookup table rules are supposed to be static and could be preloaded. It means that all rules are ready before the switch works and will not change during the classifier working. The working status of the switches also remains unchanged. Such a kind of packet classification is viewed as offline packet classification problem. The work on this problem mostly focuses on lookup performance, memory usage, and some other targets.

Meanwhile, in a typical SDN implementation, SDN devices could join and leave frequently, and SDN applications may start and stop at any time. The network is changeable, and the working status of SDN switches also dynamically changes during working. Correspondingly, this kind of packet classification is called online packet classification problem. In online packet classification problem, low latency must be taken into consideration while implementing packet classification methods. On the one hand, the update latency of the packet classification method in SDN switches influences the forward speed of the switches

when the lookup table changes. On the other hand, the latency influences the overall network performance. The SDN controllers need to wait for the reactions of the switches when they change the status of the SDN network. Many new network implementations [3] also put forward the same demands, so a packet classification method for online problems is intensively needed.

1.2. Summary and Limitations of Prior Art

The vast majority of work on packet classification can be divided into two major categories: the architectural and the algorithmic. Architectural approaches are based on special hardware platforms, such as Ternary Content Addressable Memory (TCAM) [4–7]. The approaches inevitably have the limitations brought by the hardware platforms as well. Generally, the approaches based on TCAM are expensive, area-inefficient, and power-hungry, which are also the inherent limitations of TCAM hardware.

The performance of algorithmic approaches depends on their algorithms, while each algorithm has its limitations. For instance, the decomposition methods, represented by Bit Vector (BV) [8–10], lack scalability, and only perform well in small sets. The partition methods [11–14], represented by decision-tree, lack of flexibility as the decision-tree is designed to serve static data. Some other methods, such as Tuple Space Search (TSS) [15] and trie methods [16,17], also have their shortcomings.

Since the trade-off among classification speed, memory footprint, and update complexity is inevitable, a good online packet classification method should give consideration to both fast lookup and fast update.In online packet classifications, the lookup performance is determined not only by the lookup speed of algorithms but also by the update speed. For example, when a flow modifying control message is sent to an SDN switch, the switch should stop forwarding and update its flow tables. As shown in Figure 1, the update requests will interrupt the processing progress of lookup requests, and as a result, the number of lookup requests the switch could finish will be less. If the update process takes too much time, the lookup requests will accumulate and the overall performance of the switch will be affected.

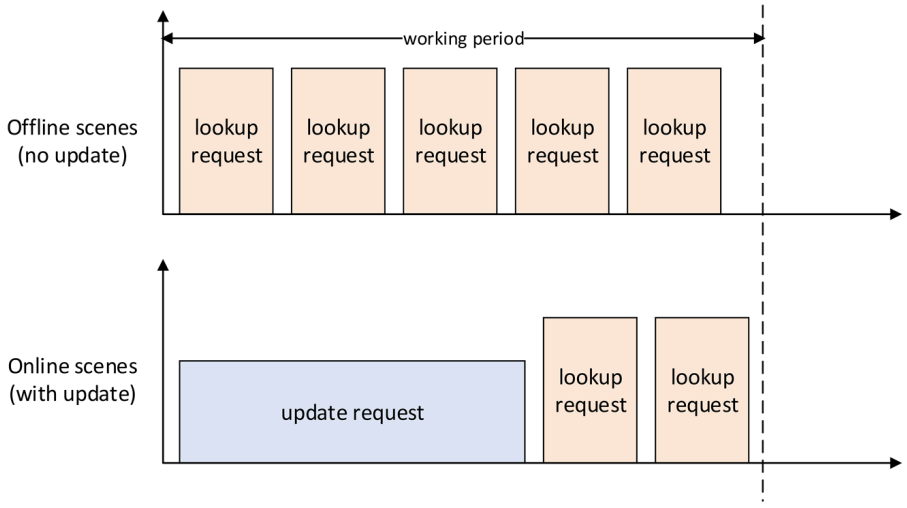


Figure 1. The two types of work scenes.

1.3. Technical Challenges and Proposed Approach

Our method is aimed to solve the online packet classification problem and provide a good packet classification algorithm. In online packet classification problems, we meet these challenges.

- Unpredictable new rule arriving time, rule contents, and arrival frequency;
- Lack of available optimization for the new work scene;
- The demand for both low update latency and high lookup speed.

It is worth noting that although the algorithms need a good update performance, the lookup speed of algorithms is still the most important in online packet classification problem. Considering this, we design our algorithm based on trie structures, which have a good lookup performance and optimize the update process. The contribution of our work can be summarized as follows:

- We put forward an online packet classification algorithm which has a good lookup performance;
- We design an update time prediction using rule wildness and trie status;
- We implement a trie choosing process based on update time predictions to avoid the potential high update latency.

The rest of this paper is organized as follows. Section 2 provides a review of the previous and related work. Section 3 presents our method in detail. Section 4 shows the experimental results to evaluate our work. Finally, this paper is concluded in Section 5.

2. Related Work

2.1. Online Packet Classification Algorithm

As we mentioned in the introduction, the work scene of online packet classification algorithms is quite different from the offline ones. So the online algorithms, which are developed from offline algorithms, changed a lot compared with the offline ones.

In fact, most work on online packet classification algorithm is the algorithmic. For instance, some online decomposition methods, such as *a packet classification algorithm with incremental updates* (PCIU) [8], design an incremental update process to get new bit-vectors when the ruleset changes. Owing to the update process, PCIU has a good update performance. However, it has the same shortcoming of poor scalability as other BV methods, even though it has good lookup performance.

For decision-tree methods, a new method called CutSplit considered update latency. It separates rules into subsets and builds trees in each subset. When the ruleset changes, it only needs to rebuild a few of the trees. PartitionSort [18] is a method that first partition the rules into some sortable subsets and then build a structure called multidimensional interval tree to support search, insertion, and deletion. These methods have similar ideas. They divide the search space and build their structures in each subset.

As TSS already meets the update requirement of online packet classification problems, the only shortcoming to be solved is the lookup speed. In TSS, the lookup speed is related to the number of tuples. TupleMerge [19] reduced the number of tuples by merging similar tuples.

Although the online methods have good update performance, most of them have not been implemented in any actual environment, and some methods only provide test codes that only support 5-tuple test benchmarks, which are quite different from our SDN environment. In the SDN implementation environment, we use a southbound interface called protocol-oblivious forwarding (POF) [20] and the match fields could be longer and more complex. So we study the optimization ideas of the online methods and design our method based on trie structure.

2.2. Classification Based on Trie

Similarly to decision tree structure, trie structure is widely used in packet classification, especially in longest prefix matching. Trie structure is a kind of storage structure, which means that the content of rules is saved in the nodes of tries. A popular trie-based packet classification instance is the Access Control List library (ACL library) provided by Data Plane Development Kit (DPDK) [21]. DPDK is a widely used set of packet processing libraries and drivers, and DPDK ACL library applies an offline packet classification method. In DPDK ACL library, the data structure build-up process has the following steps.

First, the method gets all the contents of rules in the ruleset. Then the method optimizes the ruleset. After the optimizations, the method transforms every rule into a single trie. The transformed tries then are merged into a total trie. When all tries of rules

are merged into the total trie, an array is generated from the total trie, and the array is the result of DPDK ACL buildup process. Figure 2 shows the brief buildup process of DPDK ACL.

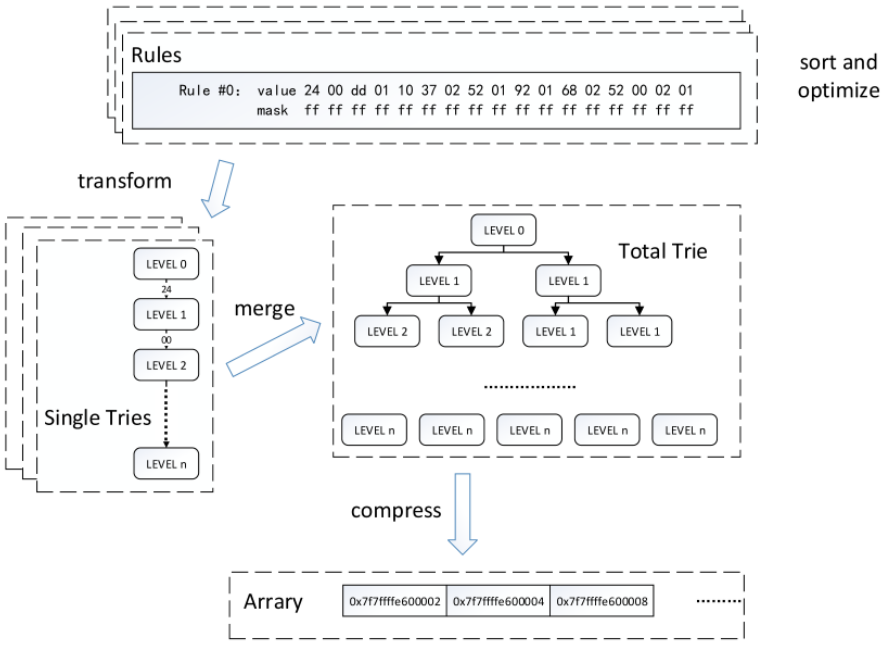


Figure 2. The buildup process of DPDK ACL.

The lookup process of DPDK ACL is quite simple. The key structure is the array generated in the buildup process. When we search in a table of trie-based methods, we only need to traverse the array and do some simple operations, such as adding, comparing, shuffling, and data reading. The total number of operations is a fixed value, so it is easy to implement a batch lookup interface. The lookup way is the key of trie-based methods to achieve high lookup speed.

In addition to the excellent performance in ruleset lookup, DPDK ACL library has no requirement for the match fields of rulesets. So this trie-based packet classification algorithm is the choice when we implemented the software switches for the SDN environment with protocol-oblivious forwarding.

2.3. Optimizations in Trie-Based Algorithm

As the raw trie-based methods have many shortcomings, some optimizations can be applied to improve the algorithm. All the optimizations are aimed to shorten the building time. One idea is reducing trie level numbers to reduce nodes when building the trie. As the level number of the trie depends on the match fields of rules, the way to reduce the level number is to skip unused match fields when building a trie. However, in online work scenes, the new rules are uncertain. No match fields can be skipped as all the match fields could be used by the next rule.

Another idea is sorting the rules. After sorting, the method will add the rules to the trie in the sorting order. If the ruleset is complex, the order can influence the final number of nodes in the trie. However, in online work scenes, the adding order should be exactly the arriving order of new rules. As both the two optimizations are not applicable in online work scenes, we will design a new optimization for the online packet classification algorithm based on tries.

3. The Proposed Algorithm

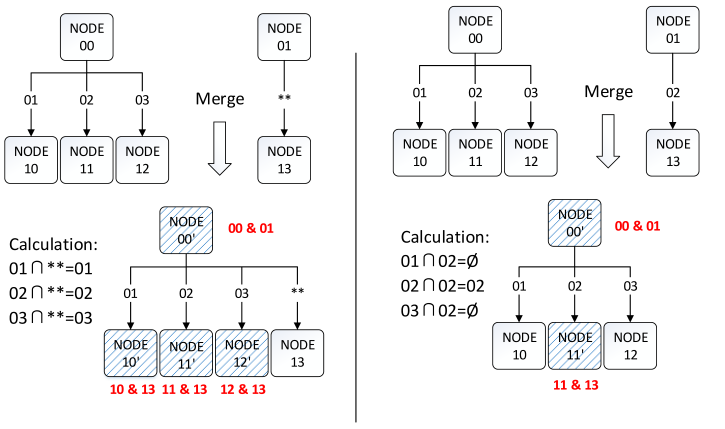
3.1. Problem and Analyzing: How to Optimize the Update Process

By updating the trie structure instead of rebuilding it, an online trie-based algorithm will take less time than DPDK ACL library to get a new lookup structure when the ruleset changes. Because in most cases, updating has a smaller time complexity than rebuilding. However, the processing time varies when the input rules are different. The update latency is uncertain as the new rule is unpredictable. Sometimes the update latency could be very high, which case is called update latency exploding.

Update latency exploding happens when a rule with many wildcards is inserted into a ruleset that already has a large number of rules. Under such circumstances, the update latency can be more than 100 times longer than usual in the worst case. To avoid update latency exploding, we should explore the update process of the online trie-based algorithm in depth.

The adding update process has only three main steps: rule transforming, trie merging, and trie compressing. The trie merging step takes the longest time. In trie merging, the nodes from the new tries will compare with other nodes in the same level of the old trie, and the node of the old trie will be operated if it intersects with the node from the new tries.

The number of nodes that will be operated in trie merging is influenced by rule content. For example, we suppose that there are two rules in a ruleset, and we mark the rules as R_a and R_b. R_a is a catch-all rule, in which all bytes of the mask are 0x00, meanwhile R_b is an exact match rule, in which all bytes of the mask are 0xff. To show the difference between the two types of rules, we consider a one-byte long merging of the match fields. The merging processes of the two rules are shown in Figure 3. When inserting R_a, all nodes in the same level intersect with the new node. When we insert R_b, only one node intersects with the new node.



a) This is a merging process of a rule with a wild byte and a trie. In This sub-figure the "**" means a wildcard byte. The rule will match the value range [0,255]. So NODE 10, NODE 11, and NODE 12 all intersect with NODE 13 .

b) This is a merging process of a rule with an exact byte and a trie. As the rule is with an exact byte, 02, only NODE 11 intersects with NODE 13.

Figure 3. The sample of merging in update process.

In addition to the type of rule content, the status of the target trie also influences the merging process time. Obviously, the number of nodes in the trie will influence the time. The more nodes the trie contains, the longer time the merging process will take. In the offline algorithm, DPDK ACL library, the method will split the trie into smaller ones when there are too many nodes in the trie.

Another trie status is the node distribution in the trie. Considering R_c is a rule which has catch-all match fields and exact match fields. If the node distribution is average in the

trie, the number of nodes to be operated will hardly change in the process of R_c insertion when the exact match fields of R_c change. However, if the node distribution is not average, the number of nodes will vary when the content of the exact match fields changes.

3.2. Update Latency Prediction

To evaluate the potential update latency, we design a latency prediction algorithm. The algorithm predicts the latency by calculating the number of potential changing nodes in the trie merging process. As we know, the trie merging process is an iteration of the node merging process, and the process will call itself several times. Each time the process calling itself, one more node will be changed. If we mark the iteration time as N and the self-calling time as A, the time complexity of the trie merging process is

$$\begin{aligned} T_m &= T_N \\ &= A_N * T_{N-1} + T \\ &= A_N * (A_{N-1} * T_{N-2} + T) + T \\ &= \sum_{i=0}^{N} \prod_{j=0}^{i} A_j * T \\ &= O(NA^N) \end{aligned} \tag{1}$$

where A is the average time of self-calling in each iteration. However, it could take a long time for us to get the excat A_j number, so we design a prediction formula. As the number A_j is determined by the new rule node content, current node number, and child node number of current nodes, so our prediction will consist of these parameters.

The first element is new rule node content. We divide the rule contents into two types: catch-all and exact match. We introduce a parameter called wildness to describe the contents. Wildness is a parameter of the match fields in rules. The wildness parameters of different types of fields are calculated in different ways. The definition of rule wildness is as following.

$$Wildness = \begin{cases} \frac{Num_0}{Num_{bit}}, & \text{for ternary field;} \\ \frac{MAX-MIN}{RANGE}, & \text{for range field;} \\ \frac{Lengh_{prefix}}{Lengh_{field}}, & \text{for prefix field.} \end{cases} \tag{2}$$

To simplify the problem, we will only consider the ternary case in the following sections, and we will treat every byte of the rule as a match field. In this paper, we will treat a byte as a catch-all byte if the wildness is bigger than 0 and treat a byte as an exact match byte if the wildness is 0. The rule content type influences the number of child nodes that will be involved in trie merging. In the trie merging process, an exact byte will involve only one child node, and a catch-all node will involve all child nodes of current nodes.

The second element is current node number. In trie merging process, current node number of this iteration is just the result of the last iteration. Therefore in the formula, we will calculate and remember the current number of each level, and the result of this level will be used in the next calculation.

The last element is the child node number of current nodes. It is easy to get the node number of each level in the trie. However, it is difficult to get the child node number of current nodes. To simplify the question, we only consider two typical child node distributions: uniform distribution and extremely skewed distribution.

Uniform distribution means that all nodes in this level have the same number of child nodes. We suppose that an empty trie is with uniform distribution of child node. With more rules added into the trie, the child node distribution becomes skew gradually. Comparing with uniform trie models, skewed trie models are more accurate but complex. We should know the skew of every node to calculate the expectation. So we first put

forward the prediction of uniform distribution case. When a rule is added to a trie with uniform distribution, the prediction value in each level is

$$F_0 = 1, F_i = \begin{cases} F_{i-1}, & W_i = 0; \\ F_{i-1} \times \frac{N_i}{N_{i-1}}, & W_i > 0. \end{cases} \tag{3}$$

N_i is the node numbers in each level of the trie, and F_i is the predicted value in the i-th level of the trie. So the total value is

$$F_{total} = \sum F_i \tag{4}$$

The actual child node distribution of the trie is not uniform generally, so when we use this formula for uniform cases to predict update latency, there will be a difference between the actual changing node number and the calculated one. When the distribution is slightly skewed, the difference is small, and we can still use the formula to predict update latency. The difference can be calculated after every update process. From the difference, we can know whether the formula for uniform cases is still suitable or not. When the old formula is no longer suitable, we will use another formula:

$$F_0 = 1, F_i = \begin{cases} F_{i-1}, & W_i = 0; \\ N_i, & W_i > 0. \end{cases} \tag{5}$$

We call Equation (3) AVE Form and call Equation (5) WC Form. The AVE Form value is an average prediction, and in most cases, the actual value is around the prediction value. The WC Form value is an upper bound. The WC Form stands for the worst case. As the skew of the nodes is big enough in the trie, it assumes that all the nodes in the next level are child nodes of the nodes which have been influenced in the merging process. It means that all the nodes in the next level will be involved in the merging process if the byte is a catch-all byte. In practice, the actual value must be less than the WC Form prediction.

3.3. Optimization: Choose a Trie

With the two prediction formulas, we design an optimization method for the trie update process. The optimization is aimed to avoid high update latency. With the merging predictions, we can know whether the latency would be high when a rule is added to a trie. Now the question is how to shorten the time when we know the time consuming would be too long.

The idea is that although we cannot decide the order of rules, we can decide which trie the rule will be inserted into. In practice, a trie-based method will build another trie when the number of nodes in the trie is too big. As a result, there would be more than one trie. So we can build several tries in the build-up process.

When we insert a rule, we choose one from the tries according to the latency predictions. For a trie, we will first use AVE Form to predict the latency. When we find the prediction is much less than the actual value, we will mark the trie as a bad trie and use WC Form to calculate latency predictions for it again. By finding a trie with a latency prediction lower than a threshold, we can avoid high update latency. Figure 4 shows the new insertion process, and the pseudocode of the new insert process is shown in Algorithm 1.

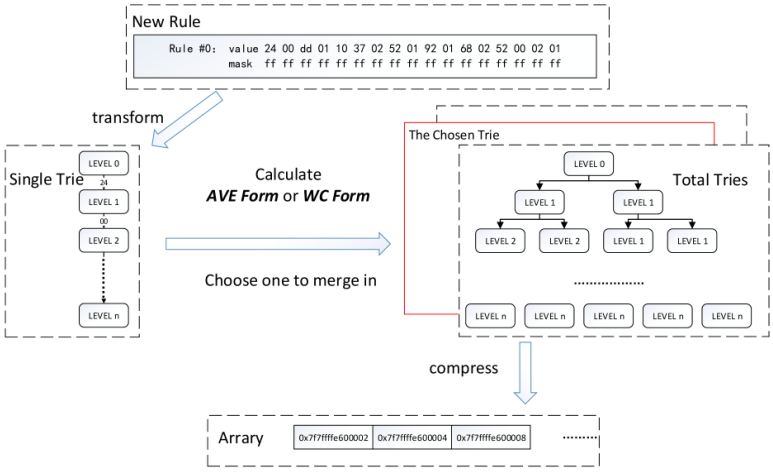


Figure 4. The update process of ITOC.

Algorithm 1: Insertion process with trie choosing.

Input: the insert rule R, ruleset $\mathbb{S}$ with tries $\{T_1\ T_2\ \ldots\ T_n\}$, threshold N_t
Output: ruleset $\mathbb{S}$ with tries

for T_i *in* $\mathbb{S}$ **do**
  if T_i *is full* **then**
    continue;
  else if T_i *is Good* **then**
    calculate prediction time N_p of R and T_i in AVE Form;
  else
    calculate prediction time N_p of R and T_i in WC Form;
  end
  if $N_p < N_t$ **then**
    choose T_i;
  end
end
if *no T is chosen* **then**
  init new trie T_{n+1} in $\mathbb{S}$;
  choose T_{n+1};
end
insert R into the chosen T_c;
generate lookup array of $\mathbb{S}$;
get actual node operation time N_a;
if $N_a < N_t$ **then**
  change T_c status into Bad;
end

Although our optimization could shorten the latency of trie merging, we also need to consider the time cost of the optimization. Luckily, the time complexity of our optimization is negligible compared with the time complexity of trie merging.

Our optimization consists of two steps: wildness calculation and prediction calculation. For wildness calculation, as the wildness is calculated for every byte individually, the time complexity T_w is:

$$T_w = O(N) \tag{6}$$

where N is the number of bytes in the rule match fields. For prediction calculation, the prediction is the sum of the prediction of each level, and we calculate the prediction for all the tries in the worst case. So the time complexity of prediction calculation T_p is:

$$T_p = kO(N) \tag{7}$$

The time complexity of our optimization is the sum of the two steps, so the time complexity of our optimization T_o is:

$$\begin{aligned} T_o &= T_w + T_p \\ &= O(N) + kO(N) \end{aligned} \tag{8}$$

Because k is a limited value, the time complexity of our optimization is O(N).

As calculated before, the time complexity of trie merging T_m is $O(NA^N)$ where A is the average time of self-calling in each iteration. As A is a value from [1, 255], the time complexity of trie merging is much bigger than our optimization. The operations in trie merging functions will take much more time than the calculations in our optimization. So we can conclude that the time cost of our optimization will not influence the update latency of the method.

Additionally, the influence of our optimization on the lookup performance is slight. Our optimization does not change the lookup process and the lookup data structure, so the only factor of lookup performance changing is the number of tries. Generally, online trie-based methods will build more trie structures than offline ones. Our optimization actively splits the trie structures, which means the possibility of more tries. The difference in trie number will lead to a gap in lookup performance. Fortunately, when the flow table is large, the number difference caused by our optimization is no longer significant.

In addition to update and lookup performance, we also consider the memory usage. As an algorithm for online packet classification problems, the method should maintain more data structures than offline methods. However, the core data structures of the algorithms are similar. So compared with DPDK ACL, ITOC will not have a significant increase in memory usage.

3.4. Implementation in DPDK

To validate our method, implementing it in DPDK is the best choice because the original method is a library provided by DPDK. There are several problems we meet when we implement the method.

The first one is the size of index groups in each leaf node. Theoretically, matching ranges of the rules could coincide, and the number of rules in a rule overlap is limitless. However, when we implement the method, we should limit the number of indices in the trie data structure with a maximum to save the memory space and to make the structure easy to implement. In our implementation, we choose the maximum number as 8.

The second one is to choose the thresholds of the insert process. In our method, there are three thresholds: the threshold for AVE Form, the threshold for WC Form, and the threshold for Form switch. In the implementation, our thresholds are chosen according to the results of simulations. The results are given in the next chapter. Finally, we set the thresholds as following: the threshold for AVE Form is 2000, the threshold for WC Form is 10,000, and the threshold for Form switch is 2000.

4. Experimental Results

4.1. Simulation Setup

The popular way to get test datasets is using the tool called ClassBench [22]. We use a work based on it called ClassBench-ng [23] because the old ClassBench is too old and no longer supported now. We use ClassBench-ng to generate 5-tuple rulesets to test our method. ClassBench-ng also provides three types of seed files: access control lists (ACL), firewalls (FW), and IP-chains (IPC)). So we now have three types of rulesets with different

properties. As the size of the ruleset is also important, we generated rulesets in three different sizes to show the influence of rule numbers.

The simulation program runs on a Linux platform with Intel Xeon Silver 4208 CPU@ 2.10 GHz and 64 GB of RAM. The operating system is CentOS release 7.9.2009. Our program works with DPDK, gets rulesets from configuration files, and reads data files to simulate network packet input.

As ITOC is an online packet classification algorithm improved from DPDK ACL, we conducted these experiments in the simulation environment. We first designed an experiment to find the best threshold set for ITOC. Then another experiment was finished to verify the effectiveness of the optimization in ITOC. Afterward, we provided the lookup performance comparison of ITOC and some recent algorithmic approaches. Finally, we compared the performance of ITOC with DPDK ACL library, especially the overall performance in online work scenes.

4.2. Simulation Results

Herein, we give out the results of the simulations using different threshold values. The simulations are aimed to find out proper thresholds for the implementation and other simulations. The results are shown in Tables 1–3.

Table 1. Performance of different AVE Form thresholds (WC Form threshold: 10,000, Form Switch threshold: 2000).

Dataset	100	200	500	800	1000	2000	3000	4000	5000
ACL	10	7	6	5	5	5	5	4	4
FW	16	11	9	8	8	7	7	9	10
IPC	11	7	7	7	7	6	9	10	11

Table 2. Performance of different WC Form thresholds (AVE Form threshold: 2000, Form Switch threshold: 2000).

Dataset	2000	3000	4000	5000	6000	7000	8000	9000	10,000
ACL	5	5	5	5	5	5	5	5	5
FW	7	7	7	7	7	7	7	7	7
IPC	6	6	6	6	6	6	6	6	6

Table 3. Performance of different Form Switch thresholds (AVE Form threshold: 2000, WC Form threshold: 10,000).

Dataset	1000	2000	3000	4000	5000	6000	7000	8000
ACL	5	5	5	5	5	5	5	5
FW	7	7	7	7	7	7	7	7
IPC	6	6	8	8	8	8	8	8

The simulation used three types of rulesets, which contain 10 k rules. The results show thatthe numbers of tries generated for the rulesets change when the thresholds change. For different types of rulesets, the trend of trie number changing is different. Considering all the datasets, we finally choose the value of AVE Form as 2000. Because 2000 is the best threshold for IPC datasets and a better threshold for FW datasets and ACL datasets. For other thresholds, things are similar. We choose the WC Form threshold as 10,000 and the Form switch threshold as 2000, respectively.

Then, we demonstrated the effectiveness of the trie choosing optimization of ITOC. We got the average update latency and average lookup time of the methods with and without optimization using three types of IPv4 5-tuple rulesets in three sizes generated by ClassBench-ng. We added the rule into the table and calculated the average update latency.

Then, we tested the lookup interface using the test packets data. The test packets data are generated from the rulesets, and in the data, every rule in the rulesets could find a packet matching itself. The results are shown in Figure 5.

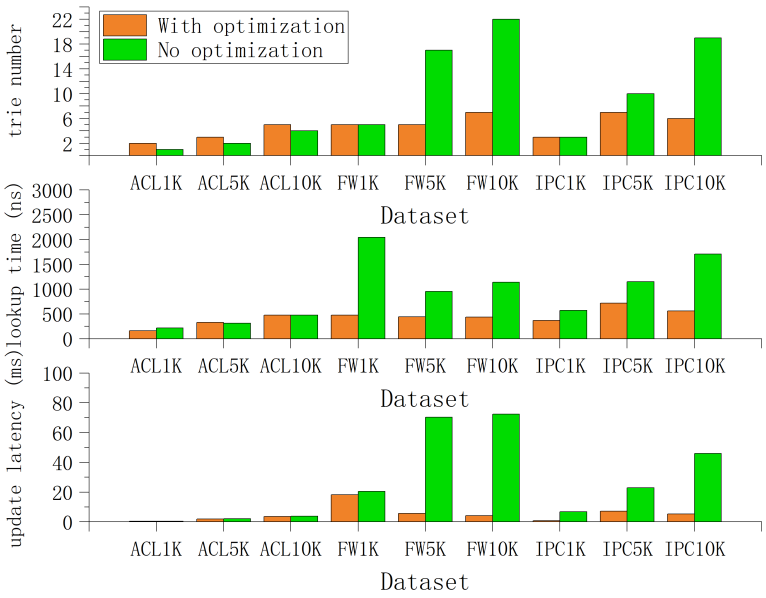


Figure 5. Performance of online trie-based method with and without optimization.

According to the results, we can find that the trie choosing optimization reduced the average update latency, especially in the FW and IPC datasets. For ACL datasets, the performance of the optimization is not as good as it for the other datasets. The larger trie number led to a longer lookup time in some experimental results. We suppose that the performance of the trie choosing optimization is related to the content of rulesets. In ACL datasets, wildcards mostly exist in source port field, and most rules in ACL datasets have an exact match source IP field. Although, in FW and IPC datasets, there are many wildcards in both IP match fields. In FW datasets, over half of the rules have IP match fields of which all mask bytes are 0x00. The differences between the datasets result in different performance of the optimization.

Then we compare the lookup performances of ITOC and some recent algorithmic approaches, represented by PartitionSort, TupleMerge, and TSS. The results are shown in Figure 6.

Compared to other algorithms, ITOC achieves the shortest average lookup time or the second shortest average lookup time in most datasets. Considering ITOC could process multiple lookup requests at once while the other algorithms could not, we believe ITOC will have more advantages in actual work scenes.

Finally, we compared the performance of ITOC with the DPDK ACL library using the IPv4 ruleset datasets. We measured memory footprint, lookup time, and update latency of ITOC and DPDK ACL. After comparing the basic performances, we simulated and compared the overall performances of the two algorithms in online work scenes. In brief, ITOC had a bigger memory footprint, the same level lookup speed, and much shorter update latency comparing with DPDK ACL. So in online work scenes, ITOC has a better overall performance than DPDK ACL algorithm.

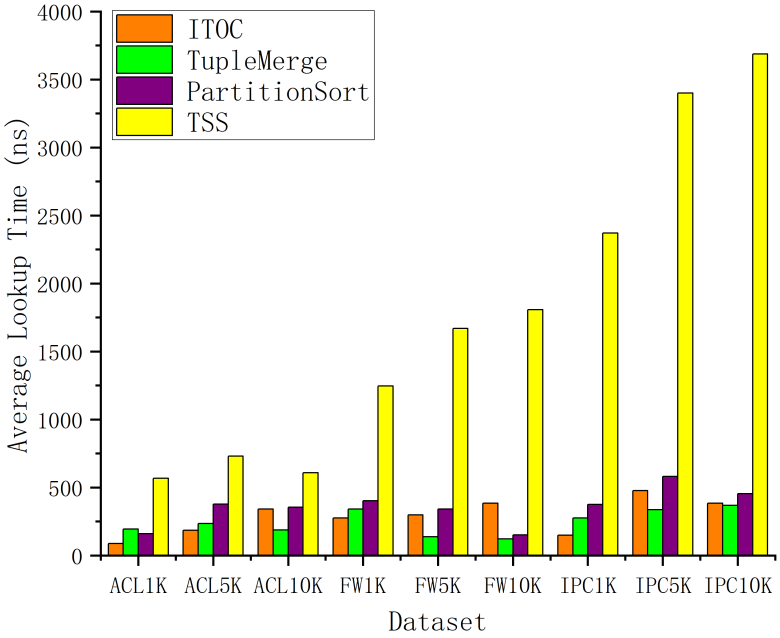


Figure 6. Lookup performance comparison.

The memory footprint comparison is as following. As the trie structures are the main reason for the memory usage, we counted the nodes in tries. The implementations of ITOC and DPDK ACL use huge page memory provided by DPDK to achieve faster memory access. So we can compare the memory footprints using the APIs. The results are shown in Figure 7.

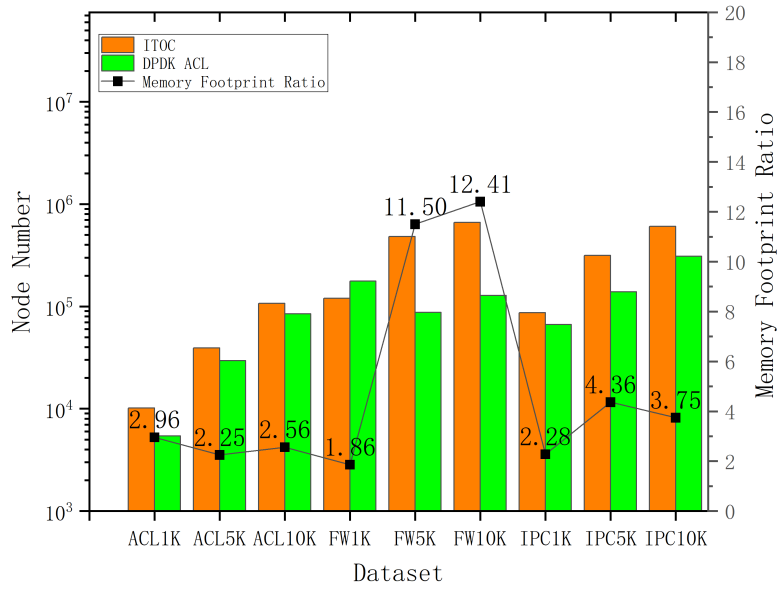


Figure 7. Node number and memory comparison.

As shown in Figure 7, ITOC has a larger memory footprint and more trie nodes. In most cases, ITOC only consumed 2 to 5 times more memory. Only in special cases, the memory footprints are bigger. ITOC consumes more memory due to the more nodes in the tries. Fortunately, since the memory is not the bottleneck of the implementation of our algorithm, the memory usage will not affect our algorithm.

The lookup performance comparison is shown in Figure 8. As both DPDK ACL and ITOC provide two lookup interfaces, we compared four interfaces in the experiment.

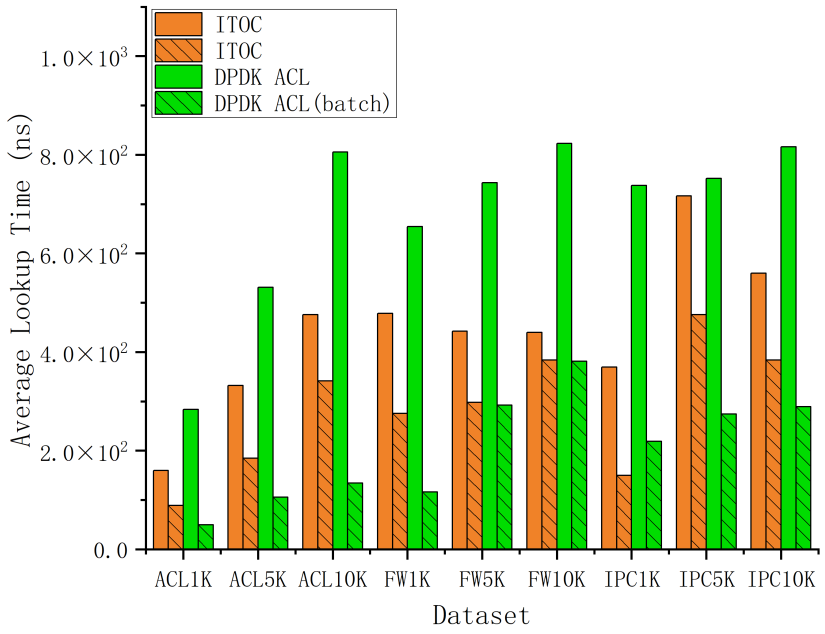


Figure 8. Average lookup time comparison.

Figure 8 shows that the batch lookup interface of DPDK ACL is the fastest lookup interface. Our online method, ITOC, has the same level of lookup speed. In some cases, the single lookup interface of ITOC is faster than DPDK ACL. Meanwhile, for batch interface, the batch lookup interface of ITOC performs poorer than DPDK ACL in most of the datasets. Briefly, In some datasets, the batch interface performance of ITOC is close to or equivalent to DPDK ACL. Considering the online work scenes, we think the performance gap from the batch interface in the other datasets is acceptable.

Then we compared the update latency of the two methods. The rules were added into tables in the same order. The results are shown in Figure 9.

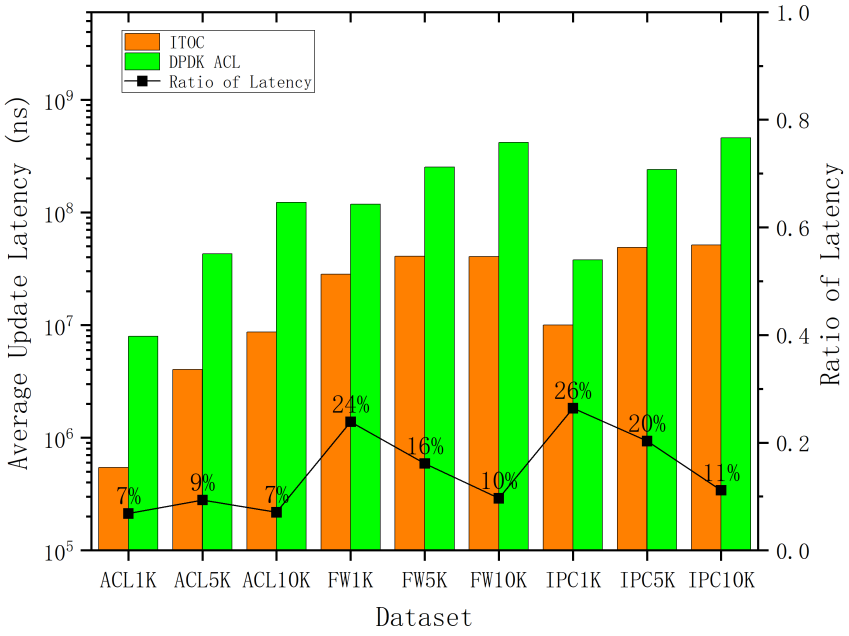


Figure 9. Average update latency comparison.

As we know, the update process of ITOC, trie updating, is much simpler than the update process of DPDK ACL, trie rebuilding. The optimization which we designed for ITOC avoided huge update latency during the update process. So in the average update latency comparison, the performance of ITOC is much better than DPDK ACL. In the best case, the update latency of ITOC is only 6.85% of DPDK ACL. The result reveals attractive advantage of ITOC on update performance over DPDK ACL.

At last, we designed a simulation to compare the overall lookup performances. In the simulation, the mixed input consists of lookup requests and update requests, which is similar to the actual online work scenes. We compared the processing time taken by each algorithm. The mixed inputs contain 10,000 requests, and the ratio of lookup requests to update requests in the inputs was set into three values: 1000:1, 100:1, and 10:1.

As shown in Figure 10, the processing time of DPDK ACL is much longer than ITOC. With the ratio of the update requests in the mixed input increasing, the processing time difference became larger. The results show that, the small sacrifice in lookup speed is acceptable as it brings considerable better lookup update performance improvement and eventually provides better overall lookup performance in online work scenes.

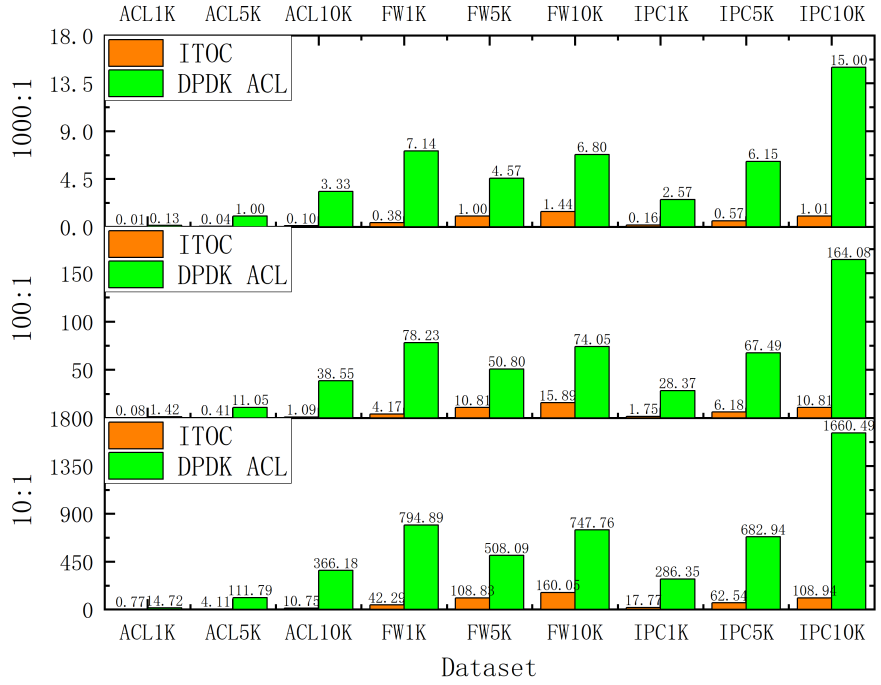


Figure 10. The mixed input processing time of ITOC and DPDK ACL(s).

5. Conclusions

Our paper is based on the contribution of DPDK ACL library and aimed to improve the performance of trie-based packet classification algorithms in online problems. In this paper, we provide the following contributions. Firstly, we design an algorithm to predict update latency for the online trie-based algorithm, and with the prediction algorithm, we could get the latency level before the update process. Secondly, based on the update latency prediction, we provide a trie choosing function to avoid update latency exploding. Finally, we implement our method and compare it with the DPDK ACL method and some recent well-known methods. In the experiment, the update latency of ITOC is only 6.85% of DPDK ACL library in the best case. Meanwhile, the lookup time of the methods is at the same level. In online work scenes simulations, ITOC achieves a better overall lookup performance than DPDK ACL. The results show that ITOC is more suitable for the SDN environment than other methods. Our future work will focus on trie update latency predictions and trie building optimization for online packet classification. Because we think there is still room for us to improve our method.

Author Contributions: Conceptualization, Y.L., J.W. (Jinlin Wang), X.C. and J.W. (Jinghong Wu); methodology,Y.L., J.W. (Jinlin Wang), X.C. and J.W. (Jinghong Wu); software, Y.L.; validation, Y.L., J.W. (Jinlin Wang), X.C. and J.W. (Jinghong Wu); writing—original draft preparation, Y.L., writing—review and editing, J.W. (Jinlin Wang), X.C. and J.W. (Jinghong Wu); visualization, Y.L.; supervision, J.W. (Jinlin Wang), X.C. and J.W. (Jinghong Wu); project administration, J.W. (Jinghong Wu); funding acquisition, J.W. (Jinlin Wang). All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by Strategic Leadership Project of Chinese Academy of Sciences: SEANET Technology Standardization Research System Development (Project No. XDC02070100).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We would like to express our gratitude to the reviewers for their helpful comments.

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

References

1. Cerović, D.; Del Piccolo, V.; Amamou, A.; Haddadou, K.; Pujolle, G. Fast packet processing: A survey. *IEEE Commun. Surv. Tutor.* **2018**, *20*, 3645–3676. [CrossRef]
2. ONF—Open Network Foundation. Available online: http:/opennetworking.org/ (accessed on 4 June 2021).
3. Wang, J.; Cheng, G.; You, J.; Sun, P. SEANet:Architecture and Technologies of an On-site, Elastic, Autonomous Network. *J. Netw. New Media* **2020**, *6*, 1–8.
4. Che, H.; Wang, Z.; Zheng, K.; Liu, B. DRES: Dynamic range encoding scheme for TCAM coprocessors. *IEEE Trans. Comput.* **2008**, *57*, 902–915. [CrossRef]
5. Liu, A.X.; Meiners, C.R.; Torng, E. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.* **2009**, *18*, 490–500. [CrossRef]
6. Meiners, C.R.; Liu, A.X.; Torng, E.; Patel, J. Split: Optimizing space, power, and throughput for TCAM-based classification. In Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, Brooklyn, NY, USA, 3–4 October 2011; pp. 200–210.
7. Kogan, K.; Nikolenko, S.I.; Rottenstreich, O.; Culhane, W.; Eugster, P. Exploiting order independence for scalable and expressive packet classification. *IEEE/ACM Trans. Netw.* **2015**, *24*, 1251–1264. [CrossRef]
8. Ahmed, O.; Areibi, S.; Fayek, D. PCIU: An efficient packet classification algorithm with an incremental update capability. In Proceedings of the 2010 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'10), Ottawa, ON, Canada, 11–14 July 2010; pp. 81–88.
9. Ganegedara, T.; Jiang, W.; Prasanna, V.K. A scalable and modular architecture for high-performance packet classification. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *25*, 1135–1144. [CrossRef]
10. Qu, Y.R.; Prasanna, V.K. High-performance and dynamically updatable packet classification engine on FPGA. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *27*, 197–209. [CrossRef]
11. Gupta, P.; McKeown, N. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro* **2000**, *20*, 34–41. [CrossRef]
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. Vamanan, B.; Voskuilen, G.; Vijaykumar, T. EffiCuts: Optimizing packet classification for memory and throughput. *ACM SIGCOMM Comput. Commun. Rev.* **2010**, *40*, 207–218. [CrossRef]
14. 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.
15. 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.
16. Fredkin, E. Trie memory. *Commun. ACM* **1960**, *3*, 490–499. [CrossRef]
17. Heinz, S.; Zobel, J.; Williams, H.E. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. (TOIS)* **2002**, *20*, 192–223. [CrossRef]
18. Yingchareonthawornchai, S.; Daly, J.; Liu, A.X.; Torng, E. A sorted partitioning approach to high-speed and fast-update OpenFlow classification. In Proceedings of the 2016 IEEE 24th International Conference on Network Protocols (ICNP), Singapore, 8–11 November 2016; pp. 1–10.

19. 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]
20. Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; pp. 127–132.
21. DPDK. Available online: http://www.dpdk.org/ (accessed on 4 June 2021).
22. Taylor, D.E.; Turner, J.S. Classbench: A packet classification benchmark. *IEEE/ACM Trans. Netw.* **2007**, *15*, 499–511. [CrossRef]
23. Matoušek, J.; Antichi, G.; Lučanskỳ, A.; Moore, A.W.; Kořenek, J. Classbench-ng: Recasting classbench after a decade of network evolution. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Beijing, China, 18–19 May 2017, pp. 204–216.