electronics

MDPI

Article

PipeCache: High Hit Rate Rule-Caching Scheme Based on Multi-Stage Cache Tables

Jialun Yang, Tao Li *, Jinli Yan, Junnan Li, Chenglong Li and Baosheng Wang

Computer College, National University of Defense Technology, Changsha 410073, China; yangjialun14@nudt.edu.cn (J.Y.); yanjinli10@nudt.edu.cn (J.Y.); lijunnan@nudt.edu.cn (J.L.); lichenglong17@nudt.edu.cn (C.L.); bswang@nudt.edu.cn (B.W.)

* Correspondence: taoli@nudt.edu.cn

Received: 24 May 2020; Accepted: 10 June 2020; Published: 15 June 2020

Abstract: OpenFlow switches hardware cannot store all the OpenFlow rules due to a limited resource. The rule-caching scheme is one of the best solutions to solve the hardware size limitation. In OpenFlow switches, Multiple Flow Tables (MFTs) provide more flexible flow control than a single table. Exact match and wildcard match are two typical matching methods. The exact match applies to a single flow table and multiple flow tables, but the performance is low under frequently changing traffic. Many commodity switches use Ternary Content Addressable Memory (TCAM) to support fast wildcard lookups. Earlier works on wildcard-match rule-caching focus on the dependency problem caused by the overlaps of match fields. Their designs cannot handle the rule-caching problem of MFTs because they are based on a single flow table, instead of the widely used MFTs structure. So, we propose a new design named PipeCache that solves the problem of MFTs rule-caching scheme based on wildcard-match. In our design, we logically split the TCAM resources, and assign them to each flow table according to the size of each flow table. Each flow table will cache the selected rules into their assigned TCAM resources and be updated in time by our algorithms to make the most use of the limited TCAM resources. We compare our structure with the exact-match scheme and the wildcard-match scheme based on a single flow table under different cache sizes and traffic localities. Experiment results show that our design PipeCache improves cache hit rate by up to 18.2% compared to the exact-match scheme and by up to 21.2% compared to the wildcard-match scheme based on a single flow table.

Keywords: software defined networking; TCAM; OpenFlow; rule caching

1. Introduction

Software-Defined Networking (SDN) is an innovative network architecture that divides the network into a control plane and a data plane and provides flexible flow control over network traffic. Different packet-processing rules manage the traffic flows in switches. These rules, based on the match-action format, according to various packet-header fields, define packets' action such as forwarding, modification, or directing to the central controller [1].

OpenFlow is a representative technology in SDN. In OpenFlow switch, match-action rules are stored in the flow tables in the form of entries. An OpenFlow switch can have Multiple Flow Tables (MFTs). OpenFlow rules as flow table entries describe how OpenFlow switch process packets [2]. Specifically, an OpenFlow rule can decide where the packet will be transferred or drop the packet according to the packet L1, L2, L3, and L4 headers, including physical switch ports, MAC addresses, IP addresses, TCP/UDP ports, and other headers. Before transferring the packet, OpenFlow Switch can change its packet header information [3–5].

In theory, all the match-actions can be handled by a single flow table. However, the combination of all the match fields causes an explosion of the flow table entries. From OpenFlow 1.1, the OpenFlow switch can set MFTs and let one packet be matched by MFTs, which provides more fine-grained and flexible flow control to switch. In OpenFlow 1.1, MFTs are organized as a pipeline and process the incoming packets successively. OpenFlow 1.1 defines the action set as a set of a series of actions corresponding to each packet. Before each packet arriving at the OpenFlow switch and matching with the MFTs, the action set will be initialized to empty. These corresponding actions will be added to the action set successively in the process of matching with the MFTs and performed at the end of the pipeline processing [6].

MFTs matching built on generic CPU allows a single OpenFlow switch to execute complex actions for a packet. Despite the flexibility of MFTs matching in software, the need for high processing speed remains challenging for generic CPU [7]. The exact matching scheme can avoid the limitation of the general CPU's performance degradation in MFTs matching. Open vSwitch (OVS) [8] and the Virtual Filtering Platform (VFP) [9] use a single hash table to store the correspondence relationship between flows and their actions instead of packet headers and actions. This scheme works well when elephant flow is the absolute dominant role in the traffic. However, when the packet fields change often, the flow caching scheme based on the exact match cannot meet the high cache hit rate requirement [7].

To avoid these shortcomings of exact match, wildcard match is now widely adopted. In the industry-standard, commodity OpenFlow switches use Ternary Content Addressable Memory (TCAM) [10,11] to store rules and perform wire-speed parallel matching. TCAM is a high-speed memory that supports parallel matching for both exact-match and wildcard-match. Compared with exact-match rules, wildcard-match rules can match more rules and improve the reusability of rules. Since TCAM is expensive, power-hungry, and takes up too much silicon space [12,13], there is only limited TCAM size in commodity OpenFlow switches, which causes the TCAM size capacity problem.

For the problem of TCAM size capacity, many solutions have been proposed. Considering the high speed and limited capacity of TCAM, many works [14–18] use the TCAM hardware flow table as a cache to store the popular rules and use the software flow table to store all of the rules. The combination of hardware and software can achieve both high-speed matching and high capacity, which has become the most popular style of OpenFlow switch. Their works focus on solving the rule dependency problem and proposed lots of different algorithms. Unfortunately, all of these wildcard-match solutions assume that there is only one flow table in an OpenFlow switch. None of them addressed the MFTs wildcard-match scheme. However, to reduce the number of rules and improve the flexibility of flow control, the MFTs have been widely used in OpenFlow switches, which remain a challenge for caching schemes based on wildcard-match. Since the existing rule caching scheme based on wildcard-match cannot be directly applied in the MFTs.

Here, we propose PipeCache to solve the problem of an MFTs rule-caching scheme based on wildcard-match. In our design, we logically split the TCAM resources and assign them to each flow table according to the size of each flow table. Each flow table will cache the most popular rules into their assigned TCAM resources. To preserve the semantics of the forwarding policies, we use the Cover-Set algorithm [15] to handle the rule dependency problem. Our structure can take the advantages of both MFTs and wildcard-match scheme and achieve the balance between high-speed matching and high capacity without causing the problem of match fields combination explosion. We compare our structure with the exact-match scheme and the wildcard-match scheme based on one flow table. Experiment results show that our design has the highest cache hit rates.

Our contributions include an innovative rule-caching scheme based on multi-stage cache tables named PipeCache and corresponding algorithms. Under different cache sizes and traffic locality conditions, PipeCache has achieved the highest cache hit rates compared with other rule-caching schemes based on MFTs. Furthermore, we organize the rest of this paper as follows. In Section 2, we introduce the background and motivation of our work. In Section 3, we present our rule-caching

scheme PipeCache. Experiment results are shown in Section 4. We describe the related works in Section 5. Section 6 is the conclusion of this paper.

2. Background and Motivation

2.1. Flow Table

In OpenFlow 1.0, packet forwarding within an OpenFlow switch is controlled by a single flow table that contains a set of entries installed by the controller [2]. The flow table entries are also called OpenFlow rules. As shown in Table 1, OpenFlow rules contain multiple fields, including match fields, actions, and a priority assigned by the controller. When a packet arrives, the switch checks it against the match fields of the OpenFlow rules. Among all the matched rules, the entry with the highest priority defines the action of the packet. The controller handles the traffic by populating OpenFlow rules in the single flow table. The single flow table is easy to implement but with lots of limitations.

Table 1. An example of an OpenFlow flow table. OpenFlow rules contain multiple fields, including match, action, counter, and priority.

Rule	Match		Priority	Action	Counter
	Field 1	Field 2			
R1	1 **	0 **	1	Forward 1	25
R2	0 **	1 **	2	Forward 2	35
R3	10 *	11*	2	Drop	15
R4	01 *	11 *	3	Forward 4	17
R5	00 *	10 *	3	Forward 5	32

* Is a wildcard character, ** Are two wildcard characters

Flexible flow control, such as performing independent actions based on matching different fields in a packet, requires a separate lookup. In theory, this can be handled in a single flow table but needs to combine all the matching fields, which can cause an explosion of the number of flow table entries. Adding more flow tables can easily handle this problem, which introduces the MFTs.

The OF1.1, published in February 2011, introduces the MFTs in OpenFlow switches. The OF1.1 supports 12 match fields, and the newly OF1.3 even supports 42 match fields, which significantly increases the flexibility of OpenFlow flow control. The MFTs are organized as a pipeline in the OpenFlow switches. This new structure brings new challenges to the OpenFlow matching methods [6].

2.2. Matching Methods

The matching methods in OpenFlow switches are an exact match and wildcard match. As mentioned in Section 1, a widely used solution for tackling the problem of generic CPU speed limitation and insufficient TCAM capacity is the caching scheme. We introduce several caching schemes based on different matching methods.

2.2.1. Exact Match

Based on the observations that in real-world network traffic, the flow distribution is skewed and has sufficient time localities to support high cache hit rates with relatively small cache sizes [7]. Due to its simplicity of implementation, many products have adopted the exact-match scheme. For example, since the first version of Open vSwitch [8], it has implemented an exact-match cache called microflow cache. The first packet of each flow will go through all of the MFTs in the "slow path". The OVS caches the matching results. The "fast path" processes the subsequent packets through a hash lookup base on the packet header fields. This mechanism ensures that only one packet of flow must go through the MFTs. The rest of the packets in this flow can rely on the fast hash lookups to avoid the slow path. When the elephant dominates the traffic, the cache hit rates are considerably high. However,

all the new flows still involve the slow path. Even the new flows require similar actions, which causes unnecessary costs. For instance, while several clients set up connections with the same HTTP server IP address and port from the same source IP address but different source ports, separate cache entries are needed [7].

Microsoft's virtual switch platform Virtual Filtering Platform VFP [9] shares the same idea of OVS. Based on the intuition that the actions for a flow are relatively stable over its lifetime, VFP caches the flow with its corresponding actions in a Unified Flow Table (UFT). The UFT is similar to the OVS microflow cache, which is also used to skip the MFTs. VFP has a unique action primitive called Header Transportation (HT) used to change or shift fields throughout a packet and a transportation engine to apply transportation to an actual packet. With the critical support of the HT and transportation engine, Microsoft tried to offload the UFT into hardware to accelerate the matching process [19]. Considering accommodating the large hash table, in their SmartNICs [19], the UFT is implemented in Dynamic Random Access Memory (DRAM) instead of the expensive TCAM. However, unlike Microsoft, most of the companies can only buy the commodity switches mostly built with TCAM. TCAM is not a cost-efficient hardware resource to implement the exact match. It is a realistic requirement to design a caching mechanism based on TCAM.

The performance of the exact-match algorithm is not dependent on the number of flow tables in the OpenFlow pipeline but the size of the cache and the flow distribution of traffic. The fact that the first packet of each flow must go through the OpenFlow pipeline is inevitable, causing the exact-match scheme's poor performance when the flows often change [7].

2.2.2. Wildcard Match

The main idea of wildcard match is to cache the popular or important rules in the TCAM of OpenFlow switches as cache entries, which is an effective way to avoid the problem of insufficient TCAM sizes. Wildcard-match scheme can maintain the wildcard rules, which makes the best use of TCAM resources.

An inevitable obstacle to the wildcard-match scheme is the rule dependency problem [16,20,21]. The neglect of rule dependency can lead to mistakes in packet forwarding. In OpenFlow, rules have four primary contents: match fields, actions, priority, and counter. Match fields come from the packet header fields. Actions define the forwarding policies of packets. For the incoming packets, the OpenFlow switch chooses the one with the highest priority in matched rules and applies the corresponding actions to the packets and increases the counter of the chosen matched rule. The priority is used to avoid conflicts when a common situation happens that one packet matches several rules but causes the rule dependency problem. If two rules overlap in their match fields, we say that the two rules have a dependency relationship, and the one with higher priority is the direct descendant of the other. In the wildcard-match scheme, if we only cache one rule without concerning its descendants, it may cause semantic forwarding errors [14–16].

An example of rule dependency is shown in Figure 1. Each rule is represented as a node in the graph. The rule R0 is the default match-all rule that can match all packets with the lowest priority, and it is added into the graph as the root node. When the rule R1 and rule R3 overlap on the match field, and the priority of R3 is higher than R1, we define that R3 is a direct descendant of R1. We add edges between all nodes that have a dependency relationship.

A naive method to solve the rule dependency problem is that when caching one rule in TCAM, we must cache all the rules that depend on it. However, this may cache lots of unpopular rules in TCAM, which wastes lots of cache entries. Lots of works have been proposed to solve this problem. We will discuss these works in Section 5. Unfortunately, all of these works on wildcard-match assume that there is only one big rule table in an OpenFlow switch. They cannot work well with MFTs. Furthermore, MFTs have been widely used in OpenFlow switches to reduce the number of rules and improve the flexibility of flow control. It is essential to design an efficient wildcard-match method that can work well with MFTs.

Rule	Match Field	Priority	Action	Counter
R1	1**	1	Forward 1	15
R2	0**	2	Forward 2	25
R3	10*	2	Forward 3	5
R4	01*	3	Forward 4	9
R5	00*	3	Forward 5	30

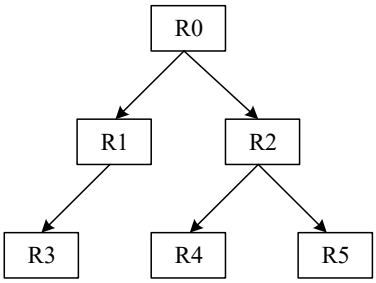


Figure 1. An example of rule dependency. Each rule is represented as a node in the graph. * Is a wildcard character. ** Are two wildcard characters.

2.3. Motivation

In theory, performing independent actions based on matching different fields in a packet can be handled in a single table. The megaflow cache algorithm of OVS [8] combines all the matched rules of each flow table into a big rule without priority as the cache. Moreover, the absence of priority speeds up packet classification and avoids the rule dependency problem [7].

For each incoming packet, it will be matched in the cache first. If it hits one rule, the switch will add the counter of the rule and execute the corresponding actions defined in the rule. If a cache miss happens, the packet will be sent to the OpenFlow pipeline and match with each flow table in the pipeline. The switch will set up a new rule, and its match fields are defined as the collection of match fields from all of the flow tables, and the action field is also the collection of actions from all of the flow tables. If the packet does not need to match one field, this field in the big rule will be set as a wildcard.

We used the traditional 12 match fields of OpenFlow as an example and assume that each rule table only contains one field. Figure 2 is an example of creating big rules in the caching table. The match fields and the actions of caching table entries are the combinations of the matched rules in the OpenFlow pipeline, which causes the caching table entries to be much bigger than those in MFTs. With the same size of TCAM, the bigger the table entries, the fewer table entries can be stored. To avoid the explosion of the number of flow entries, we only create the caching table rules with the matched rules in the OpenFlow pipeline instead of all of them.

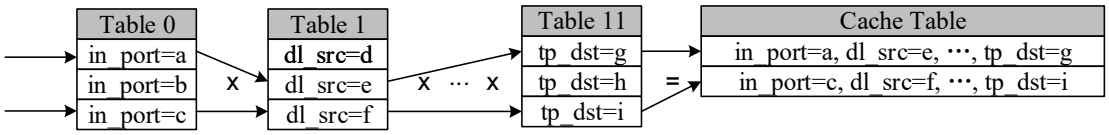


Figure 2. An example of creating big rules in the caching table. The match fields and the actions of caching table entries are the combinations of the matched rules in the OpenFlow pipeline.

Theoretically, the megaflow cache algorithm can perform well when the number of match fields is not too big, or the TCAM sizes are sufficient. Since combining too many fields and actions into one big rule and caching it in TCAM can take up lots of TCAM space, which will relatively reduce the number of cache entries and lower the cache hit rates. In PipeCache, we designed a suitable cache structure and a matching algorithm to select some of the most frequently used rules from the MFTs and cache them in high-speed hardware resources. PipeCache can preserve the correctness of forwarding and maximize the cache hit rate to improve the switch packet processing performance. As shown in Table 2, compared with PipeCache, the existing rule caching scheme either has poor performance or cannot be directly applied in the MFTs.

Table 2. Comparison between existing solutions and PipeCache.

Scheme	Matching Method	Support MFTs	Hardware Requirement	Traffic Requirement	Difficulty of Implementation	Performance
Microflow cache [8,9]	Exact	Yes	N/A	Low entropy	Easy	Low
[14–18]	Wildcard	No	TCAM	N/A	Medium	High
Megaflow cache [8]	Wildcard	Yes	TCAM	N/A	Medium	Medium
PipeCache	Wildcard	Yes	TCAM	N/A	Medium	High

3. The Design and Algorithms

In this section, we presented our MFTs wildcard-match scheme named PipeCache, which addresses the problem of generic CPU speed limitation and insufficient TCAM capacity.

3.1. Problem Statement

Flow tables are all organized as match fields, actions, priority, and counter. Each table is used to match different packet header fields. In the definition of OpenFlow 1.0, there are 12 match fields in OpenFlow rules. In OpenFlow 1.3, the number of match fields has reached 40, which makes it nearly impossible to do match-action operations in just one flow table. Many OpenFlow switches have adopted the MFTs to reduce the number of OpenFlow rules and improve the flexibility of flow control. However, if all the incoming packets are handled by the OpenFlow pipeline stored in the software and processed by generic CPUs, the processing speed will be limited. A representative way to avoid the problem is the caching scheme. By utilizing the localities of flows, we can cache the most popular flows or rules in the cache to accelerate the processing speed. TCAM is a high-speed hardware resource and suitable for the cache structure. Although TCAM can process matching in wire speed, it is an expensive and power-hungry resource. It is impossible to store all the flows or OpenFlow rules in the TCAM.

In an OpenFlow switch, the OpenFlow pipeline is stored in the software and the cache is stored in the TCAM. The incoming packets first match in cache, and then the OpenFlow switch executes the corresponding actions if the packets hit a rule in the cache. The packets will be sent to the software and go through the MFTs if packets cannot find a match in the cache. After finishing matching in the OpenFlow pipeline, the OpenFlow switch executes the actions and updates the cache entries. The matching process is shown in Figure 3.

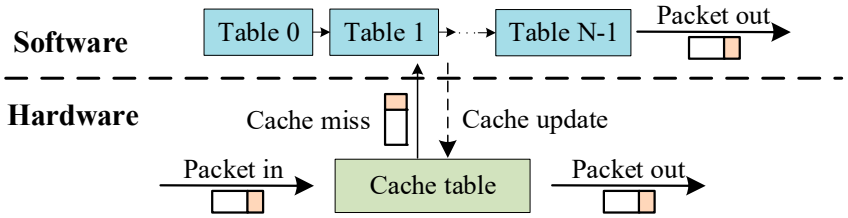


Figure 3. Matching process. The OpenFlow pipeline is stored in the software and the cache is stored in the Ternary Content Addressable Memory (TCAM).

In the exact-match scheme, the cache table's content is the correspondence between the flow ID and the action. To maximize the use of TCAM resources in commodity switches, we chose to use the wildcard-match scheme to solve the rule-caching problem of MFTs. Our goal is to design the suitable cache structure and compatible algorithms to select part of rules from the MFTs and cache them in TCAM. We want to maximize the traffic that hits TCAM while preserving the forwarding actions.

3.2. Handling Rule Dependency Problem

As we mentioned in Section 2.2, when using the wildcard-match scheme, solving the rule dependency problem is inevitable. When the match fields of two rules overlap, and we only cache the one with the lower priority, and do not cache the rule with the higher priority that overlaps the match field, a forwarding error will occur. Many works proposed their algorithms with different cache structures in different contexts. Since handling the rule dependency problem is not our key point, we can use our predecessors' great work. Considering the similar traffic environment and the simplicity of implementation, we chose the Cover-Set algorithm in CacheFlow [15].

The main idea of the Cover-Set algorithm is to build the dependency graph and break the long dependency chain to solve the rule dependency problem. Rules are regarded as nodes in the graph, and edges with arrows are regarded as dependency relationships. We traverse all the rules in the ruleset. If two rules overlap on their match fields, we add an edge between the two rules.

When caching a rule, we cannot just cache the current one in the cache but cache the one with all its direct children nodes. The action fields of these direct children nodes will be all changed to "forwarding to software", thereby avoiding caching all the long dependency chains in the cache table.

For example, in Figure 4, if R0 is in the cache table and other rules are not cached, a forwarding error occurs when a packet that should be processed according to R5 is processed according to R0 in the cache table. So, according to the dependency relationships, when caching R0, R1 to R5 all should be cached.

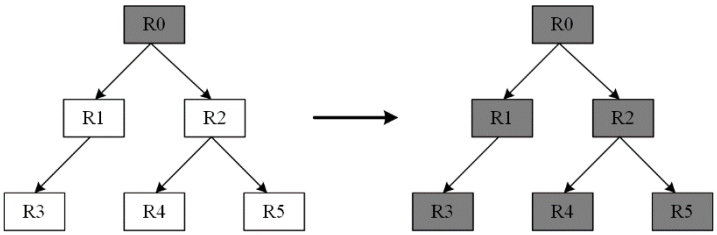


Figure 4. Rule-caching without Cover-Set.

Under the Cover-Set algorithm, after caching R0, only R1* and R2* are cached, which is shown in Figure 5. R1* and R2* have the same match fields with R1 and R2, but different action fields. The action fields of these two rules are set as "forwarding to software", thus avoiding caching R3, R4, and R5 in the cache table. When a packet hits R0 in the cache, the packet is forwarded according to R0. A packet is forwarded to software when it hits R1* or R2*. This method saves the correctness of forwarding without caching all interdependent rules and saves many cache table entries.

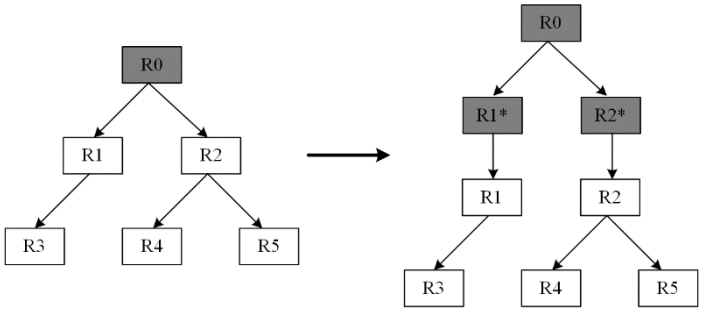


Figure 5. Rule-caching with Cover-Set.

3.3. PipeCache Strucure and Algorithm

To make the most use of MFTs and based on the fact that the TCAM resources can be formed as an arbitrary number of independent fractions, we designed the MFTs wildcard-match scheme named

PipeCache. PipeCache logically splits the TCAM resources and assigns them to these flow tables as caches. The assigned TCAM size of each flow table is based on the size of the corresponding flow table. The structure of PipeCache is shown in Figure 6.

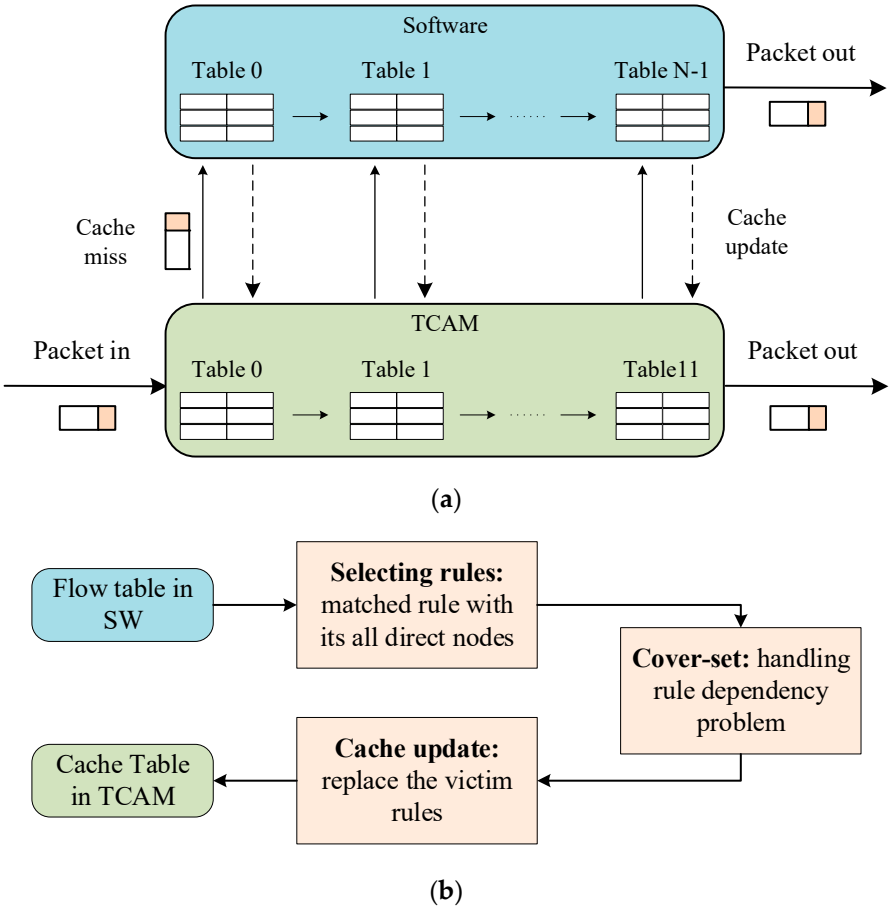


Figure 6. Overview of PipeCache. (**a**) The structure of PipeCache. (**b**) Cache update.

Although the flow tables in the OpenFlow pipeline can be formed as the combination of arbitrary match fields, we assume that each rule table only contains one match field for simplicity. The 12 match fields of OpenFlow 1.0 lead to 12 flow tables in the OpenFlow pipeline. Additionally, the form of flow tables will not influence the results of the experiments.

Figure 6a describes how the packets are processed in PipeCache. When a packet comes, it firstly matches in the MFTs in the cache in sequence. If and only if the packet hits every flow table in the cache be a cache hit. Whenever a cache miss happens, the packet will be sent to the corresponding flow table in software, and the rest of the matching process will be handled in software. Each matching in the software will lead to a cache update in TCAM. When caching a rule in TCAM, we adopt the Cover-Set algorithm to handle the rule dependency problem.

We must replace the less frequently used entries in the cache table to free up more space. Figure 6b shows the process of the cache update. We use the Cover-Set algorithm to process the matched rules in the software flow table and all direct nodes. Additionally, then, we can get the new rules to be stored in TCAM. These new rules will replace some victim rules. Many replacement algorithms [16,22–24] have been proposed. Since the replacement algorithm is not the focus of our research. Here, we use the most representative Least Recently Used (LRU) algorithm to replace the least recently used rule in the cache with the new rules in software. We can also choose another algorithm, as long as it is consistent with our comparison schemes in evaluation. In the whole process, the OpenFlow switch records the matching result of each flow table. Moreover, PipeCache executes all of the recorded actions after the

packet matching with the last flow table. The pseudo-code of the PipeCache algorithm is shown in Algorithm 1.

Algorithm 1 PipeCache algorithm

Input: p: an incoming packet;
Output: the Cache_Table _Entries in TCAM; the actions of the incoming packet;
// n: the number of MFTs;
// m: the number of rules in a rule set of a flow table;
// RS: Rule Set;
// R: Rule;
// Avail_TCAM: available TCAM entries;

```
1.    for each RS_i in cache (from RS_0 to RS_n-1) do
2.     Cover-Set (RS_i);
3.    while a packet p comes do //when a packet comes
4.      for each RS_i in cache (from RS_0 to RS_n-1) do // search in cache
5.       for each R_j in RS_i (from R_0 to R_m-1) in descending priority order: do
6.          if p matches with R_j then
7.             Add 1 to the counter of R_j;
8.             Update the LRU identifier of R_j;
9.             Continue;
10.       if p cannot find a match in Rj then
11.         for each RS_i in software (from RS_i to RS_n-1): do // search in software
12.           for each R_j in RS_i (from R_0 to R_m-1) in descending priority order: do
13.             if p matches with R_j then
14.               Set the counter of R_j to 1;
15.               Update the LRU identifier of R_j;
16.               Rules_to_cache = R_j + R_j.direct_children;
17.               if Avail_TCAM > the number of Rules_to_cache then
18.                  add Rules_to_cache in TCAM;
19.               else
20.                  delete the last recently used R for the number of Rules_to_cache times;
21.                  add Rules_to_cache in TCAM;
22.               Continue;
```

We first used the Cover-Set function to build the dependency graph for each flow table in software, and we can get each rule's direct children (Lines 1–2). The Cover-Set function is defined in Algorithm 1. Additionally, then, we start to handle the incoming packets. Each packet will be sent to the cache and match the MFTs in the cache in sequence. When the packet matches with a rule in a flow table, the OpenFlow switch will add 1 to the counter of the matched rule and update the LRU identifier, and then the packet will be sent to the next flow table (Lines 3–9). If a packet cannot find a match in the cache's flow table, it will be sent to the corresponding flow table in the software. Moreover, all the remainder matching processes will be handled by software, which is defined as a cache miss (Lines 10–12). A cache hit happens only when a packet can hit every flow table in the cache. When the packet matches with a rule in the software flow table, the switch sets the rule's counter to 1, updates the LRU identifier, and defines the rule with its direct children as the rules to cache (Lines 13–16). If there is enough space for the rules to cache, the switch directly adds them to the cache. If not, the switch will delete the last recently used rule for the number of rules to cache times before adding the rules to the cache (Lines 17–22).

4. Evaluation

In this section, we presented the simulations of our PipeCache and the strawman algorithms. We will compare our proposed algorithms with the exact-match scheme and the wildcard-match scheme based on one flow table.

4.1. Experiment Setup

Our design is based on OpenFlow rules, but we cannot get real OpenFlow switches traffic. We tried to use the widely adopted rules and traces generator ClassBench [25] to generate synthetic rule policies. However, ClassBench cannot generate OpenFlow rules and traces. Moreover, the new tool ClassBench-ng [26] can only generate OpenFlow rules, which cannot meet our requirements. So, we changed the source code of ClassBench trace generator to generate OpenFlow traces. To preserve fairness and effectiveness, we do not change the procedures and mechanisms of trace generation. The traces generated by the trace generator of ClassBench follow the Pareto distribution [25]. There are two parameters a and b in the Pareto distribution function. To fit the flow distribution of Internet traffic, the parameter a is set to 1. Additionally, the parameter b is used to adjust the locality of the generated traces. From 0 to 1, a greater value of b means higher traffic locality.

In our simulations, we use ClassBench-ng to generate multiple-stage OpenFlow rules files based on two seed files with different sizes. As shown in Table 3, for each seed file, we generate OpenFlow rule sets in three scales of 1 k, 5 k, and 10 k, and we generate ten sets of data in each scale to take the average as the results. Each OpenFlow rule has 12 match fields (Ingress port, Ethernet source address, Ethernet destination address, Ethernet type, VLAN id, VLAN priority, IP source address, IP destination address, IP protocol, ToS, transport source port, and transport destination port). We then set the locality parameter b from 10% to 20%, increasing by 2% each time to generate different trace files with different localities. Each trace file is about ten times the size of the corresponding ruleset.

Table 3. The OpenFlow rules.

Seed	Size	Number of Rules
OF1	1 k	1093
	5 k	5021
	10 k	10,381
OF2	1 k	964
	5 k	5449
	10 k	9945

Theoretically, MFTs can be formed as an equivalent big flow table. Likewise, we can split the big flow table into an equivalent pipeline of MFTs. The big flow table can be divided into combinations of arbitrary match fields. A sub flow table can be formed with single or multiple-stage match fields. Since the combinations of match fields do not influence our algorithms, we split the big flow separately by the match field for simplicity. In each flow table, we only store one entry for the same match fields. The priority of a rule is defined by its order in its flow table. The total number of rules is the sum of the number of rules in each flow table. The TCAM size is set as a parameter to evaluate the cache hit rates, which scales from 5% to 30% of the total rules number.

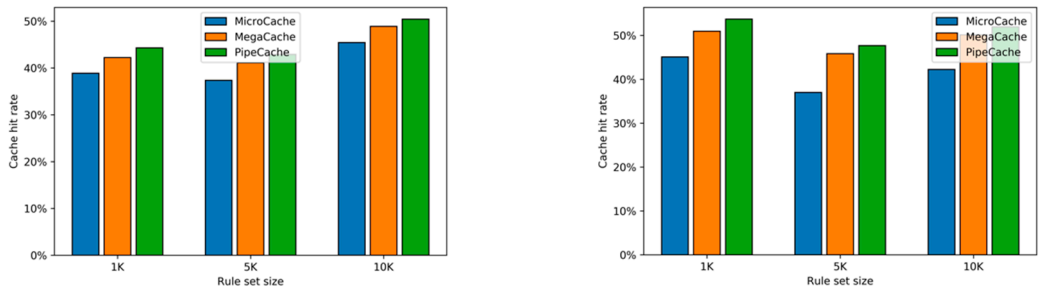
4.2. Experiment Results

To evaluate the performance of PipeCache, we compare it with the exact-match scheme and the wildcard-match scheme based on one flow table in terms of the cache hit rate. Here we call them MicroCache and MegaCache respectively for simplicity.

The cache hit rate is an important criterion to indicate cache algorithms. In the three caching schemes, TCAM is used as a cache in the OpenFlow switch. So, the size of TCAM is positively correlated with the cache hit rates. Moreover, different trace localities can lead to different test results.

We generate multiple-stage trace files based on a range of trace localities to test the relationship between the performance of different caching schemes and the size of trace localities.

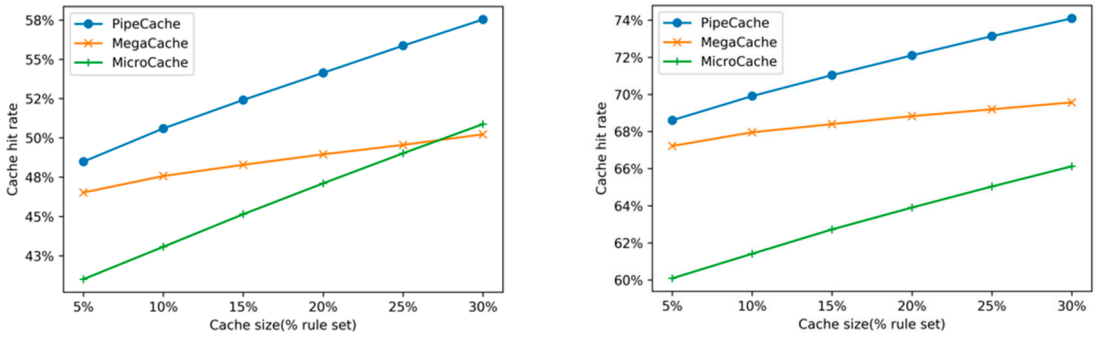
Cache hit rates with different ruleset sizes. First, we set the locality parameter b to 10% to generate the rule sets, set the TCAM size to 5% of the ruleset size, and show the cache hit rates of three schemes in different rule sets and traces. Figure 7 shows the average cache hit rates with different sizes of rule sets. As shown in this figure, our design PipeCache achieves the highest cache hit rate under both seed files while the MegaCache achieves the second-highest cache hit rates. MicroCache has the most inferior performance. When the TCAM size is 5% of the rule size, and the locality parameter b is 10%, PipeCache can outperform the MegaCache and MicroCache 4.23% and 18.25% respectively on average.



(**a**) (**b**)

Figure 7. Cache hit rates with different ruleset sizes. (**a**) OF1-seed. (**b**) OF2-seed.

Cache hit rates with different cache sizes. The size of the cache also plays an essential role in the performance of cache schemes. We set the TCAM size from 5% to 30% of the ruleset, increasing by 5% each time to test the performances of the three schemes with different cache sizes. As shown in Figure 8, the cache hit rates of the three schemes increased as the cache sizes increased. When the locality parameter was 0.1, which is a relevantly low level, the cache hit rate of PipeCache started from 48.49% with the cache size 5% and increased to 57.53% when the cache size was 30%. When the locality parameter was 0.2, the cache hit rate of PipeCache went from 68.61% to 74.09%. The other two schemes also exhibited the same trend that when the traffic had a higher locality, the caching scheme has better performance. The PipeCache outperformed the other two schemes with all cache sizes. However, when the locality parameter of traffic was 0.1, and the cache size was 30% of the ruleset, MicroCache surprisingly had a higher cache hit rate than MegaCache.



(**a**) (**b**)

Figure 8. Cache hit rates with different cache sizes. (**a**) b = 0.1. (**b**) b = 0.2.

Cache hit rates with different traffic localities. As shown in Figure 8, the change in traffic locality can influence the performance of the three schemes. To show the performance of different schemes

under different traffic localities, we set the locality parameter from 0.1 to 1.0, increasing 0.1 each time. We repeated the test three times with the cache rate at 10%, 20%, and 30%.

The results are shown in Figure 9. These figures show that the cache hit rate increased as the locality of traffic increased. Additionally, PipeCache still achieved the best performance under various traffic localities. The hit rate of MicroCache could not exceed the schemes based on wildcard matching due to the inherent shortcomings of the exact match. However, when the locality of traffic was not greater than 0.2, the performance of the MicroCache would be remarkably close to MegaCache. This is because the locality of traffic was not large enough, limiting the performance of MegaCache. When the locality of the traffic was greater than 0.2, we could see that the performance of the two schemes based on wildcard matching far exceeded the MicroCache based on exact matching. When the cache rate was not greater than 40%, PipeCache had a more significant advantage than the MegaCache. As the locality of traffic gradually increased, the performance advantage of PipeCache over MicroCache gradually decreased. The higher the traffic locality, the more popular flows will occupy a larger proportion, which also improved the caching schemes. Under a typical network environment, the locality of traffic will be maintained in a limited range, which makes the advantages of our solution PipeCache more obvious.

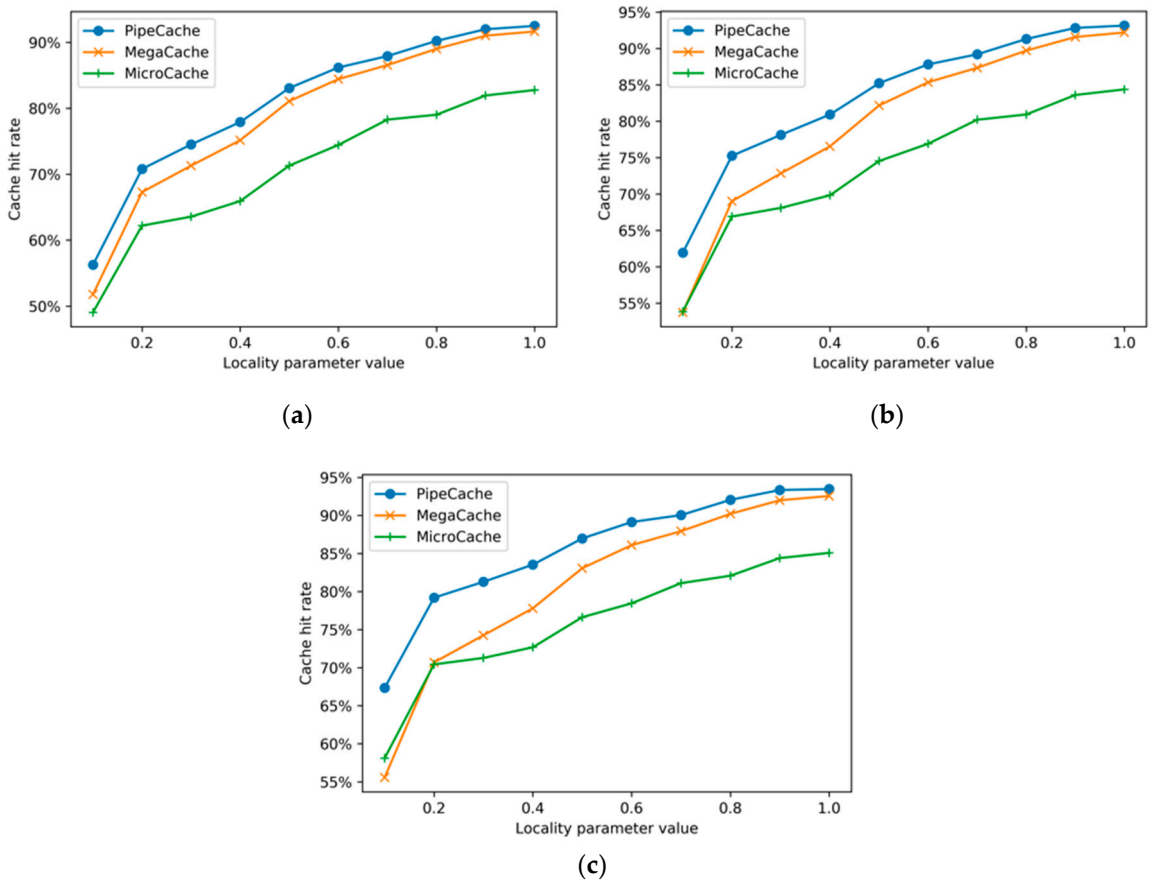


Figure 9. Cache hit rates with different traffic localities. (**a**) Cache rate = 10%. (**b**) Cache rate = 20%. (**c**) Cache rate = 30%

The results are shown in Figure 9. These figures show that the cache hit rate increased as the locality of traffic increased. Additionally, PipeCache still achieves the best performance under various traffic localities. The hit rate of MicroCache cannot exceed the schemes based on wildcard matching due to the inherent shortcomings of the exact match. However, when the locality of traffic is not greater than 0.2, the performance of the MicroCache will be remarkably close to MegaCache. This is because the locality of traffic is not large enough, limiting the performance of MegaCache. When the locality of the traffic was greater than 0.2, we can see that the performance of the two schemes based on wildcard

matching far exceeded the MicroCache based on exact matching. When the cache rate was not greater than 40%, PipeCache had a more significant advantage than the MegaCache. As the locality of traffic gradually increased, the performance advantage of PipeCache over MicroCache gradually decreased. The higher the traffic locality, the more popular flows will occupy a larger proportion, which also improves the caching schemes. Under a typical network environment, the locality of traffic will be maintained in a limited range, which makes the advantages of our solution PipeCache more obvious.

5. Related Work

Our work was related to the rule caching schemes based on wildcard-match in OpenFlow switches. We reviewed related work in this area as follows.

CAB [14] focuses on the problem of one rule may have too many direct descendants and utilizes a two-stage pipeline flow tables to solve the dependency problem. The main idea of CAB is to partition the full field space into logically independent structures named buckets, which are non-overlapping with each other, and cache buckets along with all the associated rules. CAB can not only preserve the correctness of packets forwarding but also save the control bandwidth and reduce the average flow set time.

Craft [17] take the advantages of CAB's two-stage pipeline structure and the idea of rule-partition, by adding a threshold to reduce the chances of storing multiple-stage overlapped rules in the cache to avoid generating too much rule fragments. Compared with CAB, Craft has much higher cache hit rates.

CacheFlow [15] uses a compact data structure Directed Acyclic Graph (DAG) to capture all dependencies. The authors think the long dependency chains are the main point and come up with the concept of Cover-Set to handle the dependency problem. When caching a rule, its direct descendant in DAG will be cached together, but their actions are replaced with forwarding to the software. By breaking up the long dependency chains, CacheFlow achieves high cache hit rates while maintaining the forwarding correctness.

In Ref. [16], based on the Cover-Set concept, the authors come up with a new cache replacement algorithm to improve cache hit rates. CuCa [18] proposes a design that combines the ideas of Cover-Set and rule-partition to deploy rules in a single hybrid switch efficiently.

These works all assume that there is only one flow table in an OpenFlow switch. They cannot directly be used in the MFTs pipeline structure. A naïve way to adopt these caching schemes is to combine all the MFTs into one big flow table, which can cause an explosion of the number of flow table entries like we mentioned in Section 2.

6. Conclusions and Future Work

In this paper, we proposed PipeCache, an innovative and efficient rule caching scheme designed for the OpenFlow switches with MFTs. To take the advantages of wildcard-match, PipeCache utilizes the feature that TCAM can be logically split into independent parts and creatively splits the TCAM resources in the form of a multi-stage pipeline. To make the most use of TCAM resources, we allocated TCAM to each stage of the flow table as a cache according to the size of the MFTs in the OpenFlow switch. To the best of our knowledge, this is the first attempt to adopt the rule caching scheme based on wildcard-match on the MFTs of OpenFlow switches. Experiment results show that our design PipeCache improves cache hit rate by up to 18.2% compared to the exact-match scheme and by up to 21.2% compared to the wildcard-match scheme based on a single flow table.

However, there are also some directions that suggest improvements. For example, we can improve the structure and algorithm of PipeCache so that it can be applied to the large-scale flow tables in the Wide-Area Networks [27]. Additionally, we can analyze the per-flow statistics to adaptively store the rules [28]. Moreover, many modern commodity switches have a variety of hardware resources so they can implement both exact match and wildcard match. Our future work will focus on designing a

hybrid matching method, which can take advantage of both exact match and wildcard match so that the switch can achieve the highest packet processing performance in different traffic scenarios.

Author Contributions: Conceptualization, J.Y. (Jialun Yang) and T.L.; Funding acquisition, T.L. and B.W.; Methodology, T.L.; Project administration, T.L. and B.W.; Software, J.Y. (Jialun Yang) and C.L.; Supervision, T.L. and B.W.; Writing—original draft, J.Y. (Jialun Yang); Writing—review and editing, J.Y. (Jialun Yang), J.Y. (Jinli Yan), J.L. and C.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by grants from the National Natural Science Foundation of China (No. 61802417 and No. 61702538), and the Scientific Research Program of the National University of Defense Technology (No. ZK18-03-40 and No. ZK17-03-53)

Acknowledgments: The authors would like to thank the anonymous reviewers for their valuable feedback.

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

Abbreviations

The following abbreviations are used in this manuscript:

MFTs	Multiple Flow Tables
TCAM	Ternary Content Addressable Memory
DRAM	Dynamic Random Access Memory
PipeCache	Pipeline Cache
SDN	Software-Defined Networking
OVS	Open vSwitch
VFP	Virtual Filtering Platform
UFT	Unified Flow Table
HT	Header Transportation (HT)
DAG	Directed Acyclic Graph

References

1. Kreutz, D.; Ramos, F.M.V.; Verissimo, P.; Rothenberg, C.E.; Azodolmolky, S.; Uhlig, S. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* **2014**, *103*, 16–76. [CrossRef]
2. Mckeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.L.; Rexford, J.; Shenker, S.; Turner, J.S. OpenFlow: Enabling innovation in campus networks *ACM Sigcomm Comput. Commun. Rev. 2008. 38*, 69–74.
3. Li, C.; Li, T.; Li, J.; Li, D.; Wang, B. Memory Optimization for Bit-Vector-Based Packet Classification on FPGA. *Electronics* **2019**, *8*, 1159. [CrossRef]
4. Shi, Z.; Yang, H.; Li, J.; Li, C.; Li, T.; Wang, B. MsBV: A Memory Compression Scheme For Bit-Vector-based Classification Lookup Tables. *IEEE Access* **2020**, *8*, 38673–38681. [CrossRef]
5. Li, C.; Li, T.; Li, J.; Shi, Z.; Wang, B. Enabling Packet Classification with Low Update Latency for SDN Switch on FPGA. *Sustainability* **2020**, *12*, 3068. [CrossRef]
6. Fundation, O.N. The Benefits of Multiple Flow Tables and Ttps. Technical Report, ONF Technical Report. 2015. Available online: https://www.opennetworking.org/wp-content/uploads/2014/10/TR_Multiple_Flow_Tables_and_TTPs.pdf (accessed on 11 June 2020).
7. Shelly, N.; Jackson, E.J.; Koponen, T.; Mckeown, N.; Rajahalme, J. Flow caching for high entropy packet fields. In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, Chicago, IL, USA, 17–22 August 2014; Volume 44, pp. 151–156.
8. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P.; et al. The design and implementation of open vswitch. In Proceedings of the 12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15), Oakland, CA, USA, 4–6 May 2015; pp. 117–130.
9. Firestone, D. {VFP}: A Virtual Switch Platform for Host {SDN} in the Public Cloud. In Proceedings of the 14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17), Boston, MA, USA, 27–29 March 2017; pp. 315–328.
10. Salisbury, B. TCAMs and OpenFlow-What Every SDN Practitioner Must Know. 2012. Available online: https://www.sdxcentral.com/articles/contributed/sdn-openflow-tcam-need-to-know/2012/07/ (accessed on 11 June 2020).

11. Kogan, K.; Nikolenko, S.I.; Rottenstreich, O.; Culhane, W.; Eugster, P. SAX-PAC (Scalable And eXpressive PAcket Classification). In Proceedings of the 2014 ACM Conference on SIGCOMM, Chicago, IL, USA, 17–22 August 2014.
12. Meiners, C.R.; Liu, A.X.; Torng, E. Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs. *IEEE/ACM Trans. Netw.* **2009**, *20*, 488–500. [CrossRef]
13. Cheng, Y.; Wang, P. Packet Classification Using Dynamically Generated Decision Trees. *IEEE Trans. Comput.* **2015**, *64*, 582–586. [CrossRef]
14. Yan, B.; Xu, Y.; Xing, H.; Xi, K.; Chao, H.J. CAB: A reactive wildcard rule caching system for software-defined networks. In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, Chicago, IL, USA, 17–22 August 2014; pp. 163–168.
15. Katta, N.P.K.; Alipourfard, O.; Rexford, J.; Walker, D. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In Proceedings of the SOSR 2018: The Symposium on SDN Research, Los Angeles, CA, USA, 28–29 March 2018; p. 6.
16. Sheu, J.; Chuo, Y. Wildcard Rules Caching and Cache Replacement Algorithms in Software-Defined Networking. *IEEE Trans. Netw. Serv. Manag.* **2016**, *13*, 19–29. [CrossRef]
17. Li, X.; Xie, W. CRAFT: A Cache Reduction Architecture for Flow Tables in Software-Defined Networks. In Proceedings of the 2017 IEEE Symposium on Computers and Communications (ISCC), Heraklion, Greece, 3–6 July 2017.
18. Li, R.; Pang, Y.; Zhao, J.; Wang, X. A Tale of Two (Flow) Tables: Demystifying Rule Caching in OpenFlow Switches. In Proceedings of the 48th International Conference on Parallel Processing, Kyoto, Japan, 5–8 August 2019.
19. Firestone, D.; Putnam, A.; Mundkur, S.; Chiou, D.; Dabagh, A.; Andrewartha, M.; Angepat, H.; Bhanu, V.; Caulfield, A.M.; Chung, E.S.; et al. Azure accelerated networking: SmartNICs in the public cloud. In Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation, Renton, WA, USA, 9–11 April 2018; pp. 51–64.
20. Huang, H.; Guo, S.; Li, P.; Liang, W.; Zomaya, A.Y. Cost Minimization for Rule Caching in Software Defined Networking. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 1007–1016. [CrossRef]
21. Grigoryan, G.; Liu, Y. PFCA: A programmable FIB caching architecture. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, Ithaca, NY, USA, 23–24 July 2018; pp. 97–103.
22. Megiddo, N.; Modha, D.S. Outperforming LRU with an adaptive replacement cache algorithm. *Computer* **2004**, *37*, 58–65. [CrossRef]
23. Cheng, T.; Wang, K.; Wang, L.C.; Lee, C.W. An in-switch rule caching and replacement algorithm in software defined networks. In Proceedings of the 2018 IEEE International Conference on Communications (ICC), Kansas City, MO, USA, 20–24 May 2018; pp. 1–6.
24. Lee, D.Y.; Wang, C.C.; Wu, A.Y. Bundle-updatable SRAM-based TCAM design for openflow-compliant packet processor. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **2019**, *27*, 1450–1454. [CrossRef]
25. Taylor, D.E.; Turner, J.S. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.* **2007**, *15*, 499–511. [CrossRef]
26. Matousek, J.; Antichi, G.; Lucansky, A.; Moore, A.W.; Korenek, J. ClassBench-ng: Recasting ClassBench after a Decade of Network Evolution. In Proceedings of the ACM/IEEE Symposium on Architectures for Networking & Communications Systems, Beijing, China, 18–19 May 2017.
27. Xiong, B.; Wu, R.; Zhao, J.; Wang, J. Efficient Differentiated Storage Architecture for Large-Scale Flow Tables in Software-Defined Wide-Area Networks. *IEEE Access* **2019**, *7*, 141193–141208. [CrossRef]
28. Bera, S.; Misra, S.; Jamalipour, A. Flowstat: Adaptive flow-rule placement for per-flow statistics in SDN. *IEEE J. Sel. Areas Commun.* **2019**, *37*, 530–539. [CrossRef]