Abstraction and Optimization: Applying Multiple Flow Tables in Software Defined Networks

Xiaorui Dong, Yukun Zhu
University of Toronto
xiaorui.dong@mail.utoronto.ca, yukun@cs.utoronto.edu

Abstract

1. Introduction

The emergence of Software Defined Networks (SDN)[6] eases the pain in managing networks by having a logically-centralized controller and distributed, configurable flow-based switches. SDN controllers have a global view of the entire network, while applications running on controllers can allocate and optimize the network resources by installing/removing particular rules, which specify *patterns* to match certain packets and *actions* to manage the matched packets (such as forward, drop, or send to controller). Those rules are typically stored in high-speed memories such as Ternary CAMs (TCAM). Unfortunately, commercial switches can only be equipped with limited high-speed memories due to their high manufacturing cost and high power consumption. In fact, most commercial switches today can only support up to tens of thousands of rules[8, 10].

However, the number of rules increases rapidly with the growing complexity of networks. In addition, the ultimate success of SDN and enabling technologies such as OpenFlow[5] motivate programmers to develop various applications to implement, enrich and extend the functionality of networks. Those customized applications might call for more delicate rules to exhaustively fine-tune the entire network, thus the total number of rules might easily go beyond the physical limit of switch resources. OpenFlow specification manages flow table by using per-rule inactivity timeouts to evict rules when they have not been recently accessed[14]. But it also increases the burden of controllers to install previously evicted rules once necessary.

Researchers have been well aware of the limitations of on switch memories. They have proposed various techniques to select and compress flow-table entries to lower switches memory utilization[1, 9, 13]. Novel flow table structures and multiple flow tables are also proposed to efficiently manage rules[8]. Inspired by those exciting researches, we would like to design multiple flow tables by combining both high-speed memory units (such as TCAM) and high-capacity memory units (i.e. SRAM). We can further optimize the rule placement in multiple flow tables by analysing the network model. Meanwhile, motivated by how modern operation systems manage virtual memories for each process, we would also like to provide programmers with an abstraction of switch memories. This abstraction will make our multiple flow tables completely transparent for high-level applications.

In the remainder of the report, we first review related work in Section 2. In Section 3, we introduce our abstraction for multiple flow tables and explain how we address potential issues in our architecture. Section 4 gives detailed analysis on the optimization of rule placement in multiple flow tables. Experimental results are illustrated in Section 5.

2. Related Work

2.1. TCAM and SRAM

Ternary content addressable memory (TCAM) is designed specifically for high-speed lookups. Searching in TCAM generally takes a constant time regardless the size of its content, making TCAM an ideal candidate for storing rules in network routers and switches.

However, TCAM does have several disadvantages. TCAM is less dense than SRAM. Even an area-optimized, CMOS-based TCAM cell is over 90 times larger than a DRAM cell at the same technology node, which limits the capacity of commercially available TCAM to a few megabytes[3]. Meanwhile, the comparators circuitry in TCAM cell adds complexity to the TCAM architecture. The extra logic and capacitive loading due to the massive parallelism lengthen the access time of TCAM, which is 3.3 times longer than the SRAM access time[2]. TCAM also consumes significantly more power in both read and write operations [4]. Furthermore, TCAM is not subjected to the intense commercial competition found in the RAM market. The cost of TCAM is about 30 times more per bit of storage than SRAM [11]. In contrast, RAM is available in a wider variety of sizes and flavours, is more generic and widely available, and enables to avoid the heavy licensing and royalty costs charged by some CAM vendors[12].

2.2. Optimization on Flow Tables

Existing studies on OpenFlow rule selection and placement focus on storing the most important rules on special switches, which reduces the workload on controllers. DIFANE[13] caches the most important ones at authority switches, thus that ingress switches will redirect packets which cannot match any rules in current flow table towards authority switches. Similarly, vCRIB[7] puts rules in both hypervisors and switches.

OpenFlow flow table, although flexible in structure, is not hardware friendly, which in turn impacts its scalability. Researchers from Fujitsu[8] found that commodity switches might contain exact matching tables for the forwarding databases. In order to utilize switch hardware resources efficiently, they propose DomainFlow. DomainFlow solves the limitation by splitting network into two domains. Domain 1 contains the wild-carded matching rules, and domain 2 holds the exact matching rules. DomainFlow significantly reduces the number of entries in the flow table in certain network configurations by using exact matching wherever possible. Also, splitting network into sections helps increase the usage of exact matches.

Another interesting study aims at maximizing the value of the traffic from the actual dimensioning of the network[9]. In this research, they proposed a model with several constraints that can be used to compute the best allocation of rules. Their model is applied to switches with one flow table stored on TCAM, and discards any low-weighted rules once the flow table is fully occupied.

Our project is inspired by previous researches, but also has a set of important differences. First, our model targets on combination of flow-tables which provides larger capacity for rules in general network configurations. A preliminary idea is that we combine fast, expensive TCAM with slower, cheaper SRAM to build hybrid multiple flow tables. The challenge of implementing this idea lies in how to resolve the conflict among rules generated by various applications, as well as how to address rule priorities. Moreover, we leverage the power of centralized controllers to better allocate those rules. Close scrutiny on corresponding rules in distributed switches might be critical to achieve this goal. Finally, our model should also be orthogonal with certain previous methods [13, 1] owing to abstraction —those methods should run on top of our method as if there is only one flow table per switch.

3. Abstraction for Conflict-free Multiple Flow Tables

The benefit of multiple flow tables lies mainly in their large capacity, but it also brings several drawbacks. The threshold issue for concatenated multiple flow tables is their considerable latency if we exhaustively search all flow tables to find the exact rules for a particular incoming flow. Several methods such as pure divide-and-conquer could effectively reduce the search domain for rule matching, but those methods usually rely on a particular structural of flow tables arrangement, and are less likely to preserve line-speed processing in certain circumstances. Meanwhile, tuning such methods on a given network is often artificial and labour-intensive.

Another drawback for multiple flow tables is their unfriendly programming interface for network programmers. Assuming we have a perfect system that could intelligently find the right flow table for any incoming flows, will programmers know the details of such operation? Ideally, this perfect system should run transparently from a programmer's view to enable scalability and portability. But transparency also reduces the ability for this system to adapt to different applications and network conditions.

In this paper, our proposed model addresses these two issues by providing a universal strategy to find the best match in multiple flow tables. Meanwhile, our model does take input from programmers —this enables our method to better optimize itself for a particular application. But we also hide all implementation details of our method from programmers to ensure the scalability and portability of our method.

3.1. Abstractions for Two Flow Tables

We start introducing our method with a simple example, where each switch is equipped with two flow tables. Without loss of generality, we assume one flow table is implemented with fast, expensive TCAM and the other is implemented with large, slow SRAM. For TCAM we have a fixed capacity of c_t and its latency is τ_t while for SRAM we have c_s and τ_s. We also define matching pattern m, which is a finite set of packet headers and represents the matching information (i.e. a combination of source IP address, destination IP address and port number) for each rule $r = \{m_r, a_r, s_r, p_r\}$, where m_r is the matching pattern of this rule, a_r is its action, s_r represents collected statistics and p_r is its priority. Our switch will match the header h of each incoming flow to a list of rules $\{r\}$, find the matched rule r^* with highest priority and execute the corresponding action a_{r^*}. This process can be modelled as a matching function $S \in h \times \{r\} \rightarrow \{\emptyset, a\}$, where h and $\{r\}$ is a particular header and a set of rules, respectively, and a is the final action after matching the all the rules in $\{r\}$. $S(h, r) = \emptyset$ if and only if $\forall r \in \{r\}, h \notin m_r$. In this paper we generally ignore the statistic field s_r as it doesn't have impact on our abstraction and optimization.

Assuming both flow tables are full and the frequency of visiting each rule is the same, the lowest average latency for this two flow table architecture is $\frac{c_s\tau_s+c_t\tau_t}{c_s+c_t}$. Greedily searching the entire flow table space for the best match will result in a space utilization of 100% and latency of $\tau_t+\tau_s$ for every incoming flow. Divide-and-conquer will manually set a partition $\{H_t, H_s\}$ from the hyperspace of h, thus that a minimum number of rules are split by $\{H_t, H_s\}$. This partition can be obtained by solving the minimal cut problem on a undirected graph. Any flow with header $h \in H_t$ will trigger lookup in TCAM and that with $h \in H_s$ will search in SRAM. Statistically, this technique will have average latency of $\tau_p + \frac{c_s\tau_s+c_t\tau_t}{c_s+c_t}$, where τ_p is the latency in finding the right partition for each flow header. However, finding the right partition that maps each h to $\{H_t, H_s\}$ is computational expensive, and it will also heavily impact the space utilization of this method. Moreover, any rule whose matching pattern m is in both $\{H_t, H_s\}$, that is, any rule that crosses the partition, will be kept in both flow tables. This in turn increases the total size of memory required.

Ideally, our system should keep high space utilization and comparatively low average latency. Inspired by the divide-and-conquer approach, our method adopts a partition in rule spaces $\{R_t, R_s\}$, that is, any rule that belongs R_t should be stored in TCAM and that in R_s is in SRAM. Assuming rules in $\{R_t\}$ and R_s are *independent*, which means the union of all matching patterns of rules in R_t and R_s have empty intersection, then we could search TCAM first to find possible matches. If not found, we then search SRAM and return corresponding action for a particular packet header. If no match is found in both tables, we divert the packet to the controller. In this scheme, we will have an average latency of $\frac{c_t\tau_t+c_s(\tau_s+\tau_t)}{c_s+c_t}$. Consider that $\tau_t \ll \tau_s$, the latency of our system is close to the optimal $\frac{c_s\tau_s+c_t\tau_t}{c_s+c_t}$. Meanwhile, our system could also reach the space utilization of 100%.

Although this system looks nice, but we have made an assumption that rules in R_t and R_s are independent. This independence ensures that any packet header h will only match either the rules in R_t or R_s. It is very likely that such partition doesn't exist in practice. However, the key constraint of our method is to ensure any match found in TCAM will be final, and this constraint can be satisfied by finding a *conflict-free* partition in rule space. The precision definition of conflict-free partition is given in Def. 1.

Definition 1 (conflict-free) *An ordered partition in rule space $\langle R_1, R_2\rangle$ is conflict-free if and only if for any packet header h, $S(h, R_1) = \emptyset$ or $S(h, R_1) = S(h, R_1 \cup R_2)$*

Based on Def. 1, an arbitrary rule set R will always have at least two conflict-free partition, $\langle R, \emptyset\rangle$ and $\langle \emptyset, R\rangle$. But these partitions are trivial in implementing our multiple flow tables. Unless otherwise specified, in the rest of this paper we regard all conflict-free partition as *non-trivial conflict-free partition*, which ensures $R_1, R_2 \neq \emptyset$.

Another issue of conflict-free partition is that if there will be at least one non-trivial conflict-free partition for a given rule set. Unfortunately, Table. 1 illustrates such an example.

h	$\{r\}$	$S(h,\{r\})$
h_1	$\{r_1\}$	a_1
h_1	$\{r_2\}$	a_2
h_1	$\{r_1, r_2\}$	a_1
h_2	$\{r_1\}$	a_3
h_2	$\{r_2\}$	a_2
h_2	$\{r_1, r_2\}$	a_2

Table 1: An example where no conflict-free partition exists.

It is easy to prove that no conflict-free partition exists for rule set $\{r_1, r_2\}$ with the given matching function S in Table. 1 (the only two non-trivial partition $\langle\{r_1\},\{r_2\}\rangle$ and $\langle\{r_2\},\{r_1\}\rangle$ are not conflict-free). This example also demonstrates that seeking a conflict-free partition requires knowledge on matching functions for each packet header. Luckily, those matching functions must satisfy a few constraints listed in OpenFlow standard[5], that is, those functions must be uniquely determined by the matching pattern, action and priority of each rule. That means for each packet header h, the set of matched rules $R_m = \{r | S(h,\{r\}) \neq \emptyset\}$ will either be $\emptyset$ (no match found for this packet header, divert to controller by default) or have a rule $r^* \in R_m$ such that $\forall r \in R_m, r \neq r^*, p_{r^*} > p_r$. The matching function in previous case will be $S(h, R) = \emptyset$ and that in latter case is $S(h, R) = a_{r^*}$, where R is the set of all rules in flow tables. We define the term *stable rule set* for such rule sets as in Def. 2.

Definition 2 (stable rule set) *A rule set R is a stable rule set if and only if $\forall h$, either $\forall r \in R, h \notin m_r$ or $\exists r^* \in R$, such that $\forall r' \in R, h \in m_{r'}, p_{r^*} > p_{r'}$.*

The property of stable rule set casts several constraints on the matching function S in all single-valued function in $h \times \{r\}$. In this paper, we also safely assert all the rules in a given switch or router should always form a stable rule set. However, will there always exist a conflict-free partition for any stable rule set? Will the conflict-free partition for the stable rule set to be unique? How to find a conflict-free partition in all possible partitions (the total number of partitions increases exponentially with the number of rules)? We will address those issues by constructing a conflict-free partition $\langle R_1, R_2\rangle$ on a given stable rule set R.

We first define *overlap* ($\oplus$) as a binary relationship between arbitrarily two rules r_X and r_Y in Eqn. 1.

$$\oplus := \{(r_X, r_Y) | m_{r_X} \cap m_{r_Y} \neq \emptyset \wedge \forall h \in m_{r_X} \cap m_{r_Y}, \\ S(h, \{r_X\}) \neq S(h, \{r_Y\})\} \quad (1)$$

In Eqn. 1, any rules that overlap will have a non-empty intersection in their matching pattern and different action. We further define another binary relationship *stronger than* ($>$) as in Eqn. 2

$$>:= \{(r_X, r_Y) | r_X \oplus r_Y \wedge p_X > p_Y\} \quad (2)$$

The binary relationship $>$ is asymmetric but not transitive. We then propose a greedy forward algorithm to construct the conflict-free partition $\langle R_1, R_2 \rangle$.

```
Data: a stable rule set $\{R\} = \{r_1, r_2, \cdots, r_n\}$,
      desired size of $|R_1| = C_{R_1}$
Result: a conflict-free partition
      $\langle R_1, R_2 \rangle, R_1, R_2 \neq \emptyset, R_1 \cup R_2 = R$
init: $R_u = R$, ordered set $R'_1 = \emptyset$, set $R_2 = \emptyset$ ;
while $R_u \neq \emptyset$ do
    randomly pick $r \in R_u$, $R_u := R_u - r$;
    try finding the first $r' \in R'_1$ thus that $r > r'$;
    if no such $r' \in R'_1$ then
        $R'_1 := \{R'_1, r\}$;
    else
        insert $r$ in front of $r'$;
    end
    if $|R'_1| > C_{R_1}$ then
        move the last element in $R'_1$ to $R_2$;
    end
end
```

Algorithm 1: Greedy forward algorithm for constructing conflict-free partition

Multi-thread
online update

3.2. Abstractions for Multiple Flow Tables

4. Optimization

4.1. Optimization with Local Cues

4.2. Optimization with Global Information

4.3. Automatic Optimization on Weight Factors

5. Experiment

6. Conclusion

References

[1] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[2] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. *Networking, IEEE/ACM Transactions on*, 14(2):397–409, 2006.

[3] Q. Guo, X. Guo, Y. Bai, and E. Ipek. A resistive tcam accelerator for data-intensive computing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 339–350. ACM, 2011.

[4] P. Mahoney, Y. Savaria, G. Bois, and P. Plante. Parallel hashing memories: an alternative to content addressable memories. In *IEEE-NEWCAS Conference, 2005. The 3rd International*, pages 223–226. IEEE, 2005.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[6] M. Mendonca, B. A. A. Nunes, X.-N. Nguyen, K. Obraczka, and T. Turletti. A survey of software-defined networking: past, present, and future of programmable networks. *hal-00825087*, 2013.

[7] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Vcrib: Virtualized rule management in the cloud. In *Proc. NSDI*, 2013.

[8] Y. Nakagawa, K. Hyoudou, C. Lee, S. Kobayashi, O. Shiraki, and T. Shimizu. Domainflow: Practical flow management method using multiple flow tables in commodity switches. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 399–404. ACM, 2013.

[9] X. N. Nguyen, D. Saucez, C. Barakat, T. Thierry, et al. Optimizing rules placement in openflow networks: trading routing for better efficiency. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2014)*, 2014.

[10] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. Past: Scalable ethernet for data centers. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 49–60. ACM, 2012.

[11] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[12] Z. Ullah, M. K. Jaiswal, and R. C. Cheung. Z-tcam: An sram-based architecture for tcam. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on.*

[13] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[14] A. Zarek, Y. Ganjali, and D. Lie. Openflow timeouts demystified. *Univ. of Toronto, Toronto, Ontario, Canada*, 2012.