Analysis on Performance and Cost of Incorporating SRAM Cached Flow Table in Software-defined Network Switches

Meng Wei, Yunhao Mao, Sajad Shirali-Shahreza, Yashar Ganjali

August 22, 2019

1 Abstract

SDN switches using protocols such as OpenFlow often store flow table of flow rules in fast but expensive memory called TCAM(Ternary Content Addressable Memory). With network flows becoming much larger concurrently in modern networks, simply increasing the size of TCAM is becoming too expensive.

The Openflow protocol defines timeout for rules in flow table, however previous studies suggested that the default timeout is often too long and lack ability to adaptive for different network behaviour because reducing the timeout statically will also decase the flow table look-up hit rate and resulting more slow controller queries. Our goal is to decease the TCAM flow table occupancy without sacrificing performance of flow table.

We propose using a SRAM cache in addition to traditional flow table to solve this problem while SRAM is slower but less expensive then TCAM. By placing evicted rules from flow table into the SRAM, flow table misses would be much faster then querying the controller. By using different cache placing algorithms, at most, we saw a decrease in the cost is by 85% with cached algorithm compared to normal timeout.

2 Introduction

Software Defined Networking (SDN) is becoming increasingly popular in today's Internet infrastructures such as data centers, campus networks and many more. By separating the control plane and data plane, SDN has several benefits over a traditional network architecture including scalability, elasticity, and programmability[4].

SDN protocol such as OpenFlow uses flow rules to match packets from same flow to achieve

forwarding in switch level instead of traditional routing table. However when a completely new packet arrives to a SDN switch, it need to query the SDN controller to insert a new flow rule for that packet. The flow rules are saved in switch as a Flow Table in case new packets from the same flow arrives, so that it does not need to suffer the long latency of querying the controller[5].

In practice, Flow Tables are often stored in the expensive but fast memory called Ternary Content Addressable Memory (TCAM) that are usually priced at $350 USD per Mb and uses 15Watt/1 Mbit[7]. A typical OpenFlow switch could support from around 750 flow rules(NEC PF5820 switch) to 1500 flow rules(HP5406zl switch)[2] while current data centers can have more than 10000 concurrent unique flows a second[6]. This brings the question of how to find the balance in the performance of switch forwarding and the size of TCAM because that is not feasible to indefinitely increase TCAM sizes due to price and power consumption.

In real life, every rules in the flow table have a long fixed timeout from seconds even tens of seconds, for example, 5 seconds in Floodight, 10 seconds in POX, and some up to 60 seconds[3]. However in previous studies, it had been discovered that a 63% of flows have only 1 packet an 98.7% flows last less than 1 second in the trace file(UNIV 1) we tested[3][6]. Therefore, reducing the timeout for the flow table so that flow rules will persist for a much short time can reduce the memory consumption of flow table. However, this brings the problem of prematurely eviction of long lasting flow rules which reduce the hit rate of flow table look up and thus decreases the performance.

In order to solve this problem, we proposed a method of adding SRAM cache in SDN switch in order to offload some long-living but under-used flows to the slower but less expensive SRAM cache without sacrificing performance using different rule installation policies. Due to the nature of the hardware TCAM has a match time of O(1) and SRAM needs to search the table.

As the goal of this paper is to determine how adding cache will affect the cost and performance, there are 3 main contributions been made:

1. We provided an analytical model to map hit rate of TCAM flow table and SRAM cache to performance in terms of latency. We also created a model to measure the cost based on TCAM and SRAM sizes in terms of price.

2. We proposed 4 algorithms for placing the rule in main TCAM flow table and secondary SRAM cache. We called these algorithms as rule installation policies.

3. We implemented a simulator that mimic the behavior of SDN switch that measures the hit rate and flow occupancy to investigate the effect of policies proposed in 2, and compared them to policies with no cache involved. Finally, we performed experiments using real world data center packet trace[8] on the simulator to evaluate the impact of each of the rules.

3 Previous Studies

There were many studies attempted to reduce the flow tables sizes through various of means.

Relationship between timeout and hit(miss) rate was analyzed in other research[2]. They measured the timeout, table occupancy and miss rate resulting a positive linear relationship between timeout and table occupancy, and an inverted relationship between timeout and miss rate.

They proposed a method of having controller active eviction based on network condition. They used a clairvoyant algorithm as baseline and a TCP-Based Heuristic to check if a flow is ending however it was discovered that the improvement was not reliable for different data sets. Ranging from 74% of reduction in table occupancy to 8.5% of reduction with same level of miss rate. They also proposed method of converted flow table with a cache where flows are evicted through LRU, FIFO or random eviction when full with or without a timeout and discovered that miss rate was relatively poor with smaller table sizes.

SmartTime is an adaptive timeout algorithm[1], they used previous eviction statistics and TCAM utilization to assign timeouts to each flow rules dynamically. With a fixed TCAM size, SmartTime greatly reduce the misses and drops caused by TCAM been full compares to static timeouts. With normalized cost, they achieve 58% increase compare to static idle timeout policy or random eviction policy.

Futuer more, there were other approaches such as compressing the flow table that reduce the table size by 60% [9],but they are less relevant to this paper.

4 Modeling the question

In this section, we formulate a model of comparing performance and cost using TCAM only with adding a SRAM cache. More specifically, in this paper, we analyzed the latency of access to flow tables after a packet reaches a switch as the performance metric. To better emphasize the effect on cost, we did not set a fixed size TCAM so that we could focus on the theoretical memory capacity requirement to store the flow table while there was no evictions caused by flow table overflow.

4.1 Performance in terms of latency

In general, switches' total latency correlates to two factors: the delay of the switch in processing a single packet $t_{process}$, and the number of packets the switch has processed N_{packet}.

$$Latency = t_{process} * N_{packet}$$

Since different hardware or software structures may have distinct processing times, there are three categories of possible actions for a switch to conduct after receiving packet in our model: checking main flow table in TCAM for rule match, checking SRAM cache for rule match if applicable, or querying the controller as needed. We denote $t_{category}$ as the time of hardware or software takes to process a single packet. Notice that a single packet may have multiple different hardware or software processing it. For example, for every newly arrived packet, SDN should check the main flow table, the cache and finally the controller sequentially. To avoid duplicate counts, we use N_{TCAM} for packets only handled by the main TCAM flow table, N_{SRAM} for packets handled by both the main TCAM flow table and then SRAM caches, and $N_{Controller}$ for packets eventually handled by the controller. Thus, the previous formula could be rewritten as the following:

$$\begin{aligned} Latency =& N_{TCAM} * t_{TCAM} + \\ & N_{SRAM} * (t_{TCAM} + t_{SRAM}) + \\ & N_{Controller} * (t_{TCAM} + t_{SRAM} + t_{Controller}) \end{aligned} \tag{1}$$

Do note that due to the nature of SRAM, an entry look-up require iterate through the entire cache with $O(n)$, we assume linear search and use $t_{SRAM} = t_{read} * \#entries$.

The number of packets varies noticeably in practice. In order to fit in a general model, we divide both sides of (1) by the number of total packets processed by switches. In fact, $\frac{N_i}{N_{total}}$ is the hit rate of current processing method. Moreover, $\frac{N_{Controller}}{N_{total}}$ is the miss rate; thus, we denote it as MR to avoid ambiguity. The derived formula is related to distinct hit rates. We denote the new measurement of latency as $M(Latency)$

$$\begin{aligned} M(Latency) = \frac{Latency}{N_{total}} =& HR_{TCAM} * t_{TCAM} + \\ & HR_{SRAM} * (t_{TCAM} + t_{SRAM}) + \\ & MR * (t_{TCAM} + t_{SRAM} + t_{Controller}) \end{aligned} \tag{2}$$

This formula has one important property, which is very useful for simplifying equations later: since the sum of distinct numbers of packets equal to the number of total packets, we have

$$HR_{TCAM} + HR_{SRAM} + MR = 1 \tag{3}$$

For switches without caches, $HR_{SRAM} = 0$ and $t_{SRAM} = 0$. By (3), we have $HR_{TCAM} = 1 - MR$. The formula of (2) could be simplified as the following:

$$\begin{aligned} M_{no\_cache} &= (1 - MR) * t_{TCAM} + MR * (t_{TCAM} + t_{Controller}) \\ &= t_{TCAM} - MR * t_{TCAM} + MR * t_{TCAM} + MR * t_{Controller} \\ &= t_{TCAM} + MR * t_{Controller} \end{aligned} \tag{4}$$

For switches with caches, by (3), we can rewrite HR_{SRAM} as $HR_{SRAM} = 1 - HR_{TCAM} - MR$.

The formula of (2) could be simplified as the following:

$$\begin{aligned} M_{with\_cache} &= HR_{TCAM} * t_{TCAM} \\ &+ (1 - HR_{TCAM} - MR) * (t_{TCAM} + t_{SRAM}) \\ &+ MR * (t_{TCAM} + t_{SRAM} + t_{Controller}) \\ &= HR_{TCAM} * t_{TCAM} \\ &+ (t_{TCAM} + t_{SRAM}) - HR_{TCAM} * (t_{TCAM} + t_{SRAM}) - MR * (t_{TCAM} + t_{SRAM}) \\ &+ MR * (t_{TCAM} + t_{SRAM}) + MR * t_{Controller} \\ &= t_{TCAM} + t_{SRAM} - HR_{TCAM} * t_{SRAM} + MR * t_{Controller} \\ &= t_{TCAM} + (1 - HR_{TCAM}) * t_{SRAM} + MR * t_{Controller} \end{aligned} \tag{5}$$

In summary, we have the following models:

$$M_{no\_cache} = t_{TCAM} + MR * t_{controller} \tag{6}$$

$$M_{with\_cache} = t_{TCAM} + (1 - HR_{TCAM}) * t_{SRAM} + MR * t_{controller} \tag{7}$$

4.2 Cost in terms of table and cache occupancy

Since the prices of TCAM and SRAM are fixed for the sizes of the memory, for simplicity, we assume the costs are linear with respect to the main table or cache occupancy.

5 Experiments

5.1 SDN simulator

We implemented a simulation program [Link to github] that simulates the behavior an SDN switch in Python that reads packets from network trace files. The simulator uses source and destination IPs, protocol, and source and destination to define a rule like a SDN controllers and it will insert and discard rules to flow table based on the chosen polices and settings. We did not limit to the size of flow table, because the purpose of this investigation was to determine the table size needed to achieve a certain rate under a certain policies.

Before every $min(timeout, 100ms)$ interval, the simulator will inspect the flow table and cache to report current number of active flows, maximum count of flows, flow table and cache hit rates and other metrics.

5.2 Rules Installation Policies

We implemented 2 rule installation policies without utilizing the cache as baselines.

1. no_cache: The flow table has a fixed timeout for every installed rule and rule is evicted after timeout.

2. SmartTime Algorithm[1]: The switch dynamically assigns timeout for each flow based on its previous access pattern.

We implemented 3 cached policies with two of them having a fixed sized cache, and two of them having variable sized cache to measure required sizes of cache for a given optimal hit rate.

1. cache_no_timeout_fifo: The flow table has a fixed timeout and when a rule is evicted from the flow table, it is installed in the cache. The cache does not have a timeout but a fixed size and acts like a queue, when cache is full and a new rule to be installed, the first element is dequeued. This is an algorithm inspired by hybrid algorithm in *Openflow Timeouts Demystified* [2].

Algorithm 1 cache_no_timeout_random

```
1:  function CHECK_ALL_TIMEOUT
2:      for rule ∈ flow_table do
3:          if timed_out?(rule) then
4:              flow_table.remove(flow)
5:              if cache.full?() then
6:                  cache.dequeue()
7:              end if
8:              cache.enqueue(flow)
9:          end if
10:     end for
11: end function
```

2. cache_fixed_timeout: Both flow table and cache have a fixed timeout. The cache's timeout is a **multiplier** to flow table's timeout thus the cache size will be proportional to the flow table size. A rule is installed into the cache after a threshold of number of evictions from main flow table is exceeded. The controller will be responsible for keep track of numbers of times that one flow get evicted and notify the switch when the flow is inserting again. By doing this, we can determine the size of SRAM cache needed to maintain a certain hit rate.

3. cache_dynamic_timeout_last_rules: Flows rules in flow table has a fixed timeout, but the timeouts for rules in cache are per-rule based. Inspired by SmartTime algorithm[1], it uses the

difference in time between last two installations of that flow rule to decide the new timeout thus only on the third time when a flow rule got evicted, it would be put in the cache. The controller will keep track of this information and notify the switch when the flow is inserting again.

Algorithm 2 cache_dynamic_timeout_last_rules

```
1: function CHECK_ALL_TIMEOUT
2:     for rule ∈ flow_table do
3:         if timed_out?(rule) then
4:             flow_table.remove(flow)
5:             if flow.#_of_evicted > threshold then
6:                 flow_timeout = flow.last_insert_time − flow.last_evicted_time
7:                 cache.insert((flow, flow_timeout))
8:             end if
9:         end if
10:    end for
11:    for rule ∈ cache do
12:        if timed_out?(rule) then
13:            cache.remove(flow)
14:        end if
15:    end for
16: end function
```

6 Results

In this section, we presented the result of experiments in measuring the Hit Rate vs Table Occupancy/Size for all policies and draw the Hit Rate vs Table Occupancy graph.

The Hit Rate is defined by $H = \frac{\#\_hits\_flow\_table+\#\_hits\_cache}{\#\_all\_packets}$ and Table Occupancy is defined by 99 percentile of active flows at one time. We chose 99th percentile to represent the size of flow table or cache needed to be in order to achieve that level of hit rate without considering rule eviction due to full tables as we mentioned in previous sections.

We ran the experiments using tracefile UNIV 1[6], the data set was from a university data center and available online[8]. The trace file consists of 19 million packets, lasts 65 minutes(Dec 17, 2009, 11:26 - 12:31), contains 841 thousands unique flows, consists 41.1% of TCP packets and 58.6% of UDP packets[3].

For $cache\_fixed\_timeout$, we chose 1 and 11 as the **multiplier** for cache timeout. For $cache\_no\_timeout\_fifo$, we chose to set cache size to 200 and 800.

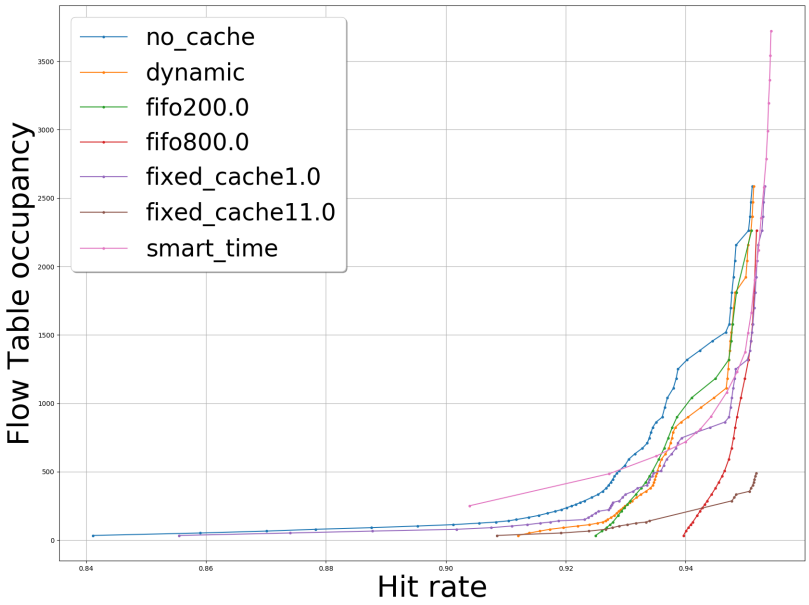

Figure 1: 99th Percentile of Flow Table Occupancy vs Hit Rate

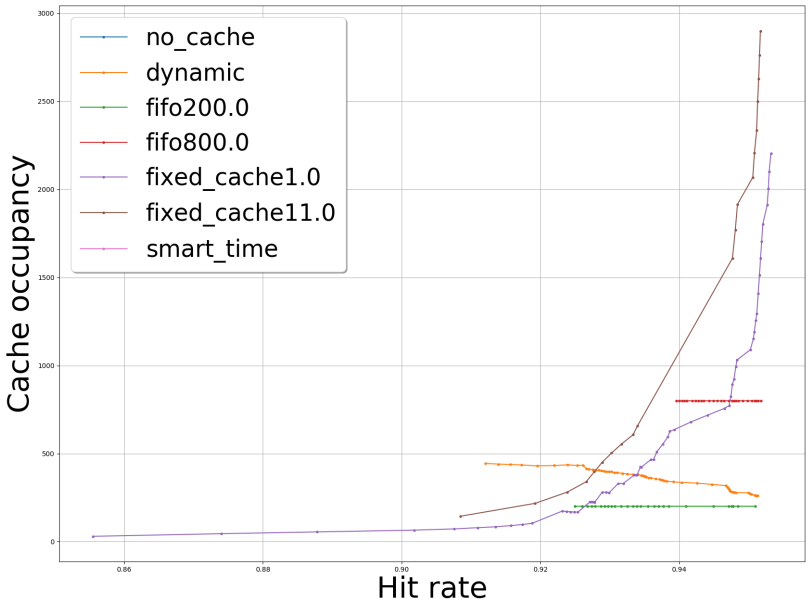

Figure 2: 99th Percentile of Cache Occupancy vs Hit Rate

One observation can be made is that cached algorithms starts off with higher hit rate even if the main flow table timeout occupancy is low, this is excepted because it essentially increase the table size. However, with the increasing of hit rate, no cached policies' memory consumption grows exponentially and the cached policies have an advantage in starting with a high hit rate.

We can observe that cache sizes for $cache\_fixed\_timeout$ grow directly proportionally toward with the flow table sizes as expected. This means that the flow table size requirement can be decrease linearly by reducing the timeout multiplier.

We could notice that $cache\_dynamic\_timeout\_last\_rules$'s cache size decreases when the flow table size increases, it requires less flow table space at lower hit rate but on higher hit rate, its cache is under utilized and require more flow table entry to achieve high hit rate.

Combining the two figures, we could see the improvement with adding a cache is exceptional. When all the flow rules can fit in the cache, a fixed sized cache beats everything else. At hit rate of 0.94, TCAM requirement for no_cache is 140 times more than $cache\_no\_timeout\_fifo$ with cache size of 800. However this may not be as cost effective if we take in account the extra price needed for the cache as we will discussed in the next section.

7 Analysis

In this section, to evaluate real life performance, we calculate the cost using the specs of NL9512 TCAM chip from Netlogic and CY7C1525 SRAM from Cypress Semiconductor[11]. A single NL9512 TCAM chip has capacity of 2.56MB and costs $390 per chip.The capacity of CY7C1525 SRAM is 9MB clocked at 250MHz and costs $89.7 per-chip. We assumed each entry is 64B[2], then the TCAM has a unit price of 0.00975 dollar per entry and the SRAM has a unit price of 0.00072 dollar per entry.

We define TCAM look-up has a latency of 5ns[12], one SRAM look up takes 5 clock cycles and with 250 MHz clock, it takes 20ns[14], and the RTT for controller query to be 10ms[2].

We are able to calculate the cost effectiveness by combining Figure 1 and Figure 2 and using the model we proposed previously by plotting Latency and Cost.

Overall average latency suffers a diminishing return when reaching lower latency. Fixed size cache policies significantly worsen when cache sized is reached as we can see that fixed size as we can see that to achieve an average latency of 0.6ms, a cache sized 800 that can hold most of rules only cost 15% of no_cache, but when trying to achieve 0.3ms of average latency, it costs similar to no_cache as expected because main table will be mostly used and dominates the cost.

If we set cache size based on the main table occupancy, the cost effectiveness is relatively well

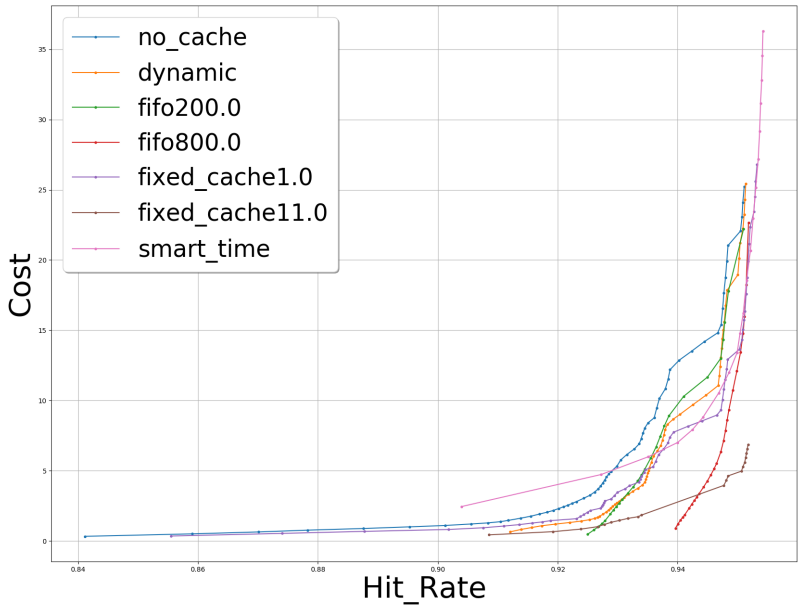

Figure 3: Cost vs Hit Rate

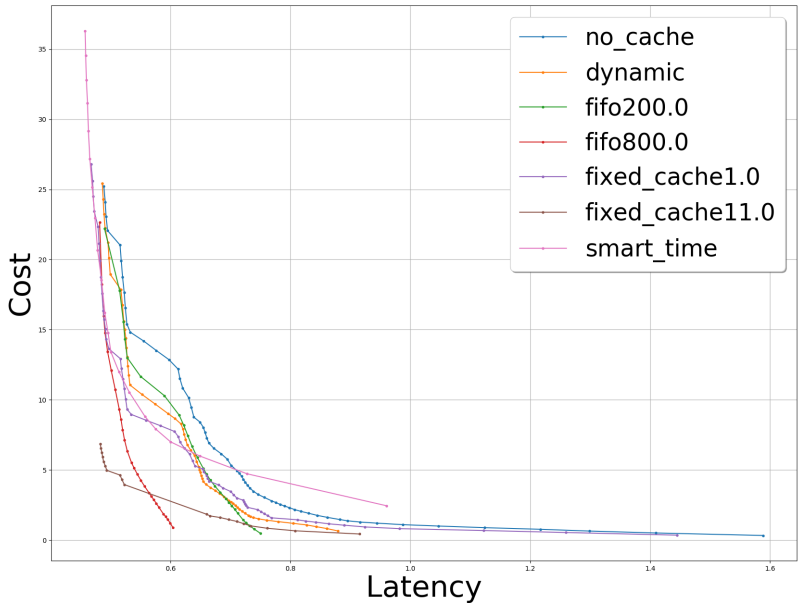

Figure 4: Cost vs Latency(ms)

comparing to all other polices at high performance region. When reaching the lower bound of latency (upper limit of hit rate), the cost of needing to hit that latency with cache, is 5 times less than smart_time will we could just have a large cache to accommodate the rules.

Therefore, we can conclude that at low performance region, the cost difference is rather insignificant where some time no_cache is slight better in cost-effectiveness. At extreme performance region, simply increases the SRAM size is cost-effective. In the middle ground, finding a suitable cache size can gain a good result. However, one thing we did not take in account in our experiment was that very large flow table, if the cache size becomes more than 10000, the search time will be more than 0.2 millisecond, which will significantly increase the overall average delay.

8 Future Works

Flow rules placing we experimented were rather naive. Comparing smart_time, cache_dynamic_timeout_last_rules and a clairvoyant algorithm from other search[2] , we observed that a good prediction of flow behaviour could significantly benefit the performance and table utilization no matter using cache or not. However the ability to predict timeout for a flow is not trivial, methods using learning algorithm with one flow suffers from lack of training data because many flows are short-lived. One improvement could achieved with analyzing the network traffic as a whole to predict the behaviour of a single flow using machine learning algorithms[13] and apply it to dynamic timeout assignment for both flow table and cache.

It has been observed that different timeout algorithms performance differently with different network applications[2]. For example, network characteristics from different data centers, or campus network vs ISP data centers are drastically different and will affect the performance of timeout policies[3]. Measuring data sets from different sources can potentially find a best policy to use for different use cases. Also, the network traffic volume is an important aspect of affecting the performance, and as we discussed before, the look-up time with SRAM will significantly larger with large cache which would need to be investigated.

9 Conclusion

In this paper, we studied the effect in performance and cost of an SDN flow table using different rules of insertion and eviction policies with the addition SRAM cache on existing TCAM storage.

We first presented a model of relationship between hit rate and performance.We then created a simulator that reads packet traces to simulate a simplified behaviour of a SDN system. We were able to use the simulator to implement few hypothetical rule installation policies that utilizing cache differently. There were total of 6 policies tested, where 2 of the policies used traditional single table system with fixed or per-flow based variable timeout, 2 of the policies used no-timeout cache with eviction on full and 2 of the policies used fixed or per-flow based variable timeout on cache. Then we conducted experiments using popular real life data center trace files.

We analyzed the results from different polices and discovered that cost-effectiveness vary at each performance level. To achieve extreme performance, adding a bigger cache can be very cost-effective because its the cheapest way to increase overall performance. For the middle ground of performance, finding a cache size that fits the number of rules. And when the performance is not a requirement, a small cache or even no cache can will be sufficient.

We believe that our study is relevant giving the increasing popularity in SDN applications.

References

[1] Vishnoi, Anilkumar, et al. "Effective switch memory management in OpenFlow networks." Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. ACM, 2014.

[2] Zarek, Adam, Y. Ganjali, and D. Lie. "Openflow timeouts demystified." Univ. of Toronto, Toronto, Ontario, Canada (2012).

[3] Shirali-Shahreza, Sajad. FleXight: Flexible Information Channel for Software-Defined Networking. Diss. 2018.

[4] Benzekki, Kamal, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. "Software-defined networking (SDN): a survey." Security and communication networks 9.18 (2016): 5803-5833.

[5] OpenFlow Switch Specification Version 1.5.1. `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`

[6] Benson, Theophilus, Aditya Akella, and David A. Maltz. "Network traffic characteristics of data centers in the wild." Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, 2010.

[7] Kannan, Kalapriya, and Subhasis Banerjee. "Compact TCAM: Flow entry compaction in TCAM for power aware SDN." International conference on distributed computing and networking. Springer, Berlin, Heidelberg, 2013.

[8] Data Set for IMC 2010 Data Center Measurement. `http://pages.cs.wisc.edu/ tbenson/IMC10_Data.html`

[9] 席孝强, 兰巨龙, 孙鹏浩, 江逸茗, 刘博. 一种保持OpenFlow功能完整性的TCAM流表压缩模型[J](Function-integral TCAM-saving compression model for flow table of OpenFlow). 《计算机应用研究》, 2018, 35(5)

[10] Metter, Christopher, et al. "Analytic model for SDN controller traffic and switch table occupancy." CNSM. 2016.

[11] Luo, Layong, et al. "Towards TCAM-based scalable virtual routers." Proceedings of the 8th international conference on Emerging networking experiments and technologies. ACM, 2012.

[12] Foysal, Md Atik, et al. "Performance analysis of ternary content addressable memory (TCAM)." Advances in Electrical Engineering (ICAEE), 2015 International Conference on. IEEE, 2015.

[13] NB Harikrishnan, KP Soman, "Detecting Ransomware using GURLS", Advances in Electronics Computers and Communications (ICAECC) 2018 Second International Conference on, pp. 1-6, 2018.

[14] Understanding Static RAM Operation

http://www.archive.ece.cmu.edu/ ece548/localcpy/sramop.pdf