An Efficient Flow Cache algorithm with Improved Fairness in Software-Defined Data Center Networks

Bu-Sung Lee[1], Renuga Kanagavelu[2], Khin Mi Mi Aung[2]
[1]EBSLEE@ntu.edu.sg , [2]{renuga_k, Mi_Mi_AUNG}@dsi.a-star.edu.sg
[1]School of Computer Engineering, Nanyang Technological University, Singapore
[2]Data Storage Institute, A*STAR (Agency for Science and Technology), Singapore

Abstract— **The use of Software-Defined Networking (SDN) with OpenFlow-enabled switches in Data Centers has received much attention from researchers and industries. One of the major issues in OpenFlow switch is the limited size of the flow table resulting in evictions of flows from the flow table. From Data Center traffic characteristics, we observe that elephant flows are very large in size (data volume) but few in numbers when compared to mice flows. Thus, Elephant flows are more likely to be evicted, due to the limited size of the switch flow table causing additional traffic to the controller. We propose a differential flow cache framework that achieves fairness and efficient cache maintenance with fast lookup and reduced cache miss ratio. The framework uses a hash-based placement and localized Least Recently Used (LRU)-based replacement mechanisms.**

Keywords — Software Defined Networking, Flow cache, Elephant flow, TCAM, Data Center

I. INTRODUCTION

Data Centers host various kinds of applications such as web service, e-commerce, and social networking which generate a large number of intra-Data Center flows. Recent studies on Data Center traffic [1] report that while 99% of the flows carry fewer than 100 Mbytes (termed usually as mice flows), while more than 90% of all the bytes are transported in flows between 100 MBytes and 1 GBytes (termed usually as elephant flows). Most of the small flows are caused by either the application keep-alive packets (ICMP) or the TCP acknowledgments. Further, applications such as MSSQL, HTTP and SMB have flow characteristics more closely related to small flows rather than large flows. In order to provide flexible and intelligent control of Data Center network traffic, flow-based networking has become an attractive solution. Software-defined networking (SDN) is a key technology for realizing Data Center network virtualization. Recently, there has been an immense interest in using OpenFlow [2] as a platform for flow-based software-defined networking.

OpenFlow defines a framework to perform per-flow routing where switches maintain flow tables. Every OpenFlow switch maintains its own table of flow entries (flow table), in which each entry contains a set of packet fields to match, and the corresponding action to perform, (e.g. forward, drop, modify header). Flow tables are built out of TCAMs that support wildcard entries and parallel lookups. The OpenFlow enabled HP 5406zl switch hardware [3] can support up to 1500 OpenFlow rule entries (OpenFlow rule is described by 10 header fields , which total 288 bits [2]) , whereas the switch can support up to 64000 forwarding entries for standard Ethernet switching (Ethernet forwarding descriptor is 60 bits(48-bit MAC +12 bit VLAN id). This is too small a number compared to the number of active flows arriving at the switch. It is possible to increase TCAM entries, but it consumes lots of ASIC space, power (about 15 Watt/1 Mbit)[4] and cost(US$350 for a 1M-bit chip) .

OpenFlow switches use per-flow timeouts to manage the lifetime of each flow rule. There are two timeout mechanisms- *idle_ timeout* and *hard_ timeout*- in OpenFlow [16]. An *idle_timeout* causes the flow entry to be removed if no packet matches the rule within a certain inactivity period. A *hard_timeout* causes the flow entry to be removed after the given number of seconds, regardless of how many packets it has matched. So, the flow entries will not stay in a flow table forever and once evicted the controller needs to reinsert the flow information into the flow table when necessary.

As stated earlier, elephant flows are very large in size (data volume) but few in numbers when compared to mice flows. Mice flooding in the Data Center network traffic will cause the elephant flows to be evicted prematurely from the flow table, particularly when, a burst of many mice flow packets arrive at a switch between elephant flow packet arrivals [14]. In [8] similar observations have been made in the Internet Traffic in the presence of small number of large Elephant flows and a burst of short Mice flows. Flow table misses cause additional delay, due to round-trip time to the controller and in addition an extra *packet-in* event generated and sent to the controller, incurring additional load on the controller. In order to mitigate this problem, we proposed a flow cache layer in between the switch and controller. For a scalable solution, caches can be used with each flow cache storing records for a number of switches. Thus a cache comprises of a number of flow caches each corresponding to a flow table of a switch. When required, a switch first consults the flow cache instead of directly contacting the controller.

To facilitate better sharing of cache memory, we propose a cache architecture wherein, a cache corresponding to a switch is organized into dynamically growing and shrinking number of blocks (or buckets) with associated indices. The size of the index depends on the current number of buckets. The index value is obtained by hashing the relevant fields of a flow,

978-1-4799-0568-3/13/$31.00 ©2013 IEEE

called dynamic-index hashing. To ensure fair treatment of elephant flows, we propose a differentiated flow cache mechanism wherein we associate different indices for elephant and mice flows, and also for the buckets. By organizing the cache into multiple buckets we keep the number of entries in a bucket that need to be searched as small as possible. We note that, the search (or lookup) operation with fewer cache entries helps reduce the lookup time when realized with SRAM. The cache entry replacement strategy used in our framework is a localized least recently used (LRU) which was shown in our experiments to be more effective than other strategies.

Our proposed differentiated caching framework has several attractive features. The cache architecture and hash function are simple, thus enabling ease of implementation. The dynamically growing/shrinking feature of the flow cache enables better sharing of the cache memory by different flow caches. The differentiated index approach improves fairness for elephant flows reducing the number of their evictions. Also, our placement, lookup, and replacement mechanisms ensure fast flow processing with high hit rate.

The rest of the paper is organized as follows. Section II presents the background and related work. Section III presents the proposed cache architecture and its maintenance operations. Section IV studies the performance of the proposed mechanisms and discusses the simulation results. We conclude the paper in Section V.

II. Back Ground and Related work

Software-defined networking (SDN) is an approach to networking in which control is decoupled from packet forwarding. OpenFlow [3] is a standard, which enables users to easily implement experimental routing protocols via software and becoming the dominant protocol for SDN. The key difference between the conventional router and the OpenFlow is that the forwarding (data plane) and routing (control plane) layers are decoupled. The entire network is centrally managed by a dedicated controller, which communicates with OpenFlow-compliant switches using the OpenFlow protocol. Every OpenFlow switch maintains its own table of flow entries (flow table), in which each entry contains a set of packet fields to match, and the corresponding action to perform, (e.g. forward, drop, modify header etc.). In the event when a switch does not find a match in the flow table, the packet is forwarded to the controller to make the routing decisions. After deciding how to route the new flow, the controller installs a new flow entry at the required switches so that the desired actions can be performed for the packets of the new flow.

One of the responsibilities of the controller is to manage the contents of the flow table. Apart from installing new rules, it is also responsible for removing flow rules. Flow tables can be built with TCAMs which are power hungry and expensive. Current switches usually support very limited number of entries, eg.1500 [3]. So, the controller assigns a timeout period to each inactive entry. When the timeout period expires, the flow table inactive entry will be evicted and the switch notifies the controller that the flow was removed.

Depending upon the lookup mechanism there can be two kinds of flow tables - exact match table and wildcard-aware linear table [6]. The exact match table enables fast lookup using hash calculation, but requires maintenance of a large number of flow entries. On the other hand, the wildcard-aware linear table can reduce the number of the entries, but requires linear search resulting in longer lookup time.

Zadnik et, al. use a genetic algorithm to summarize the feature vector of elephant flows and apply the vector for picking up subsequent heavy-hitters [7]. This approach claimed to achieve cache hit rate close to the optimal. But, the training process is expensive and requires a large traffic dataset for good performance.

Tian et, al. proposed a flow cache replacement policy based on the statistically positive correlation between the flow length of the packets and the flow cache evict times [8] to identify the elephant flows and improve the cache hit rate. They proposed Adaptive least Frequently evicted (ALFE) cache replacement policy which gives high priority to elephant flows to keep them in the cache, preventing them being flooded by the massive mice flows. It divides the cache line into three segments of different sizes and uses replacement strategies in accordance with different flow durations. Flow records are inserted into one of the three segments according to their evict times when referenced. To deal with dynamic network traffic, it adjusts the flow duration threshold values, keeping the size of the segments fixed. Different from this work, our work deals with dynamically growing/shrinking buckets according to the dynamic network traffic.

Recently, researchers have proposed a few approaches for reducing TCAM power consumption. Zane et al. proposed a bit selection and Trie-based architecture [9] in which the first level TCAM is an index to the partitions in the second level TCAM. This architecture has the drawback of complex route update. Ravikumar et al. propose a two-level pipelined architecture that reduces power consumption through prefix compaction and partitioning [10]. The size of the largest page defines the worst power consumption. So, if the largest page has a high access frequency, power consumption increases quickly. This architecture is unsuitable for bursty access patterns. Wu et al, propose an algorithm that exactly partitions the routing table and a architecture with two stages, Index-TCAM and Sub-TCAM [11] to increase the lookup throughput and reduce the power consumption.

Extendible Hashing [12] is a dynamically updateable disk-based index structure which implements a hashing scheme utilizing a directory. However, it uses the concept of global depth and local depth and can possibly use keys of different size for different blocks resulting in uneven block size and variable search speed.

DIFANE [13] presents an approach to improve flow-based networks' control plane performance. It reduces the load of the controller by exploiting a special type of switch, called authority switch that is responsible for caching some portion of the flow table. It processes all packets in the forwarding

plane. However, DIFANE does not address the issue of global visibility of flow states and statistics and also requires switches to have enough CPU resources to realize the extra control plane functionalities.

III. Caching Architecture and Maintenance

In this section, we describe our proposed flow cache architecture and various maintenance operations. The cache is used by more than one switch and as a result the cache is comprised of a number of flow caches. A switch is mapped to only one cache which avoids the need for coordination between the cache modules in a large network. A flow cache is organized as a set of fixed-size buckets. To facilitate better sharing of cache memory, the number of buckets assigned to a flow cache is dynamically chosen (subject to a predefined maximum value) depending on the requirement by the corresponding switch. For simplicity, we will only use one flow cache in our discussion in the following sections.

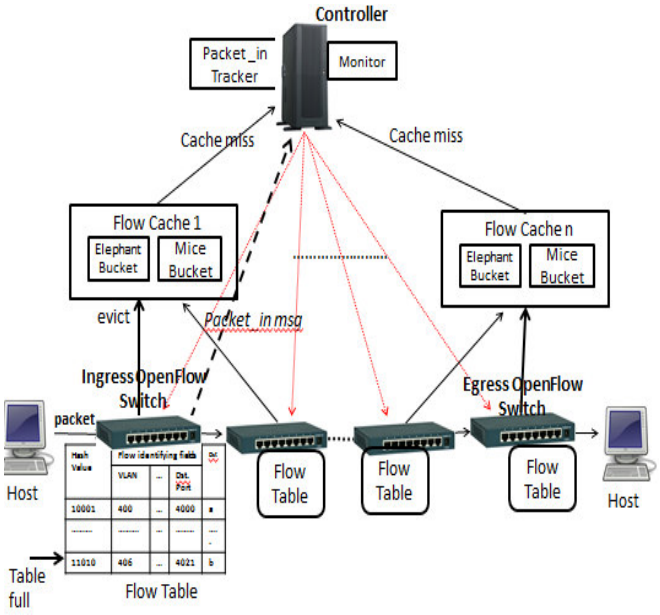

Figure 1. Interactions between the controller, cache and flow table

Figure 1 shows the interactions between OpenFlow switch, cache and the controller. An entry in the flow table may be evicted because of lack of space or based on timeout mechanism. Such entries will be stored in the associated flow cache. When a packet arrives at an OpenFlow-enabled switch, a lookup is performed first on the OpenFlow switch's flow table.

The Lookup operation is detailed in the pseudo code as shown in Figure 2. A flow table miss will result in the lookup of the flow cache; if there is a match in the flow cache, the associated action for packet forwarding will be executed; otherwise a *packet_in* event is sent to the controller leading to an insertion of an entry in the flow table. If a cache bucket is full and the maximum number of buckets has already been used up, new flow entries cause a cache miss leading to the eviction of an existing flows.

In a dynamic network environment, where the controller performs dynamic re-routing based on the real-time traffic load, the controller could change the path of the big flow to avoid congestion [17]. In that scenario, the controller will send flow *modification* messages to both flow cache as well as switch to update them.

The insertion of the flow cache layer in the architecture results in the exchange of few messages like flow_mod , packet_in messages between cache layer and the controller .

Algorithm for LOOKUP OPERATION at Flow Cache

```
Read Packet arrives at the OpenFlow switch ,
   Check the OpenFlow switch's flow table for matched entry
   If (found) then
      Perform action for packet forwarding
   else check the flow cache.
      If (cache_hit) then
         Perform action for packet Forwarding
      else
Send packet_in message to controller
      endif
   endif
```

Figure 2: Pseudo code for Lookup operation

The following sections describe the main components of our frame work.

A. *Elephant flow detection*

In our design we added two components, *Monitor* and *Packet-in Tracker* to the NOX controller. The purpose of the Monitor component is to query, consolidate and store the statistics from all OpenFlow switches. The statistics is collected from each OpenFlow switch by polling them at fixed intervals. The per-table, per-flow and per port statistics are gathered from all the connected OpenFlow switches and stored in memory as a snapshot object. Each snapshot is identified by a sequence number, which increments by 1 after each interval.

We use the flow statistics mechanism to identify the Elephant flows. Each flow is monitored at the first switch traversed by the flow. The statistics are collected from switches by the controller at a fixed interval of p seconds and used to classify the large flows. It is computed as follows:

$$\Psi_t = (b_t - b_{t-p})/p$$

Where ψ_t is the estimated flow size at time t, and b_t is the total bytes of the flow received by the switch at time t. In our implementation, p is set to 5 seconds. We classify flows as elephant flow if the flow size is equal or greater than 100KB. We follow the similar approach in Mahout [12] to use the virtual LAN priority code bit (PCB) bits (set

as 001 for elephant and 000 for Mice) to indicate the elephant flows as shown in Figure 3 .

Switch port	MAC src	MAC dst	Eth Type	VLAN	IP Sec	IP Dst	IP Port	TCP sport	TCP dport	Action
				PCB-001						

Figure 3. Using VLAN PCB bits to classify flows.

Packet-in Tracker processes the *packet_in* messages that are generated by the flow cache miss, and then inserts a flow entry in the flow table.

B. Flow cache and Dynamic-index hashing

The flow cache is organized into separate buckets for mice flows and elephant flows. While the maximum number of buckets in a flow cache is constrained, the number of buckets used by the elephant flows and mice flows are dynamically determined based on the requirement. We use dynamic-index hashing to map a flow to a bucket. The k-bit index associated with a flow is dynamic and the value of k depends on the number of buckets currently used for that type of flow (mice or elephant). Associated with each bucket there is an index which stores the flows (either mice or elephant, but not both) with the same index. We use the popular SHA hashing method [15] and extracting *x* bits (for forming the index) from the random hash value is more likely to keep the buckets balanced. Since SHA is a strong hashing method, it is highly likely that equal number of flows will be mapped to different buckets.

We use a directory to store the index values and the pointer to the corresponding bucket. The index directory is in the form of an array with at most 2^b entries for the indices with each entry storing a bucket address. The variable 'b' is called the maximum depth of the directory corresponding to the maximum permissible number of buckets. The index directory and 1-bit index buckets for mice and elephant flows are shown in Figure 1. Initially, the cache line has only two buckets termed as *Bucket A* with the index '0' for storing mice flows and *Bucket B* with the index '1' for storing elephant flows.

The placement of the flows into the flow cache is carried out in the following steps which are shown in Figure 4.

1. When a packet arrives, the hash value for the packet is calculated from its flow identifying fields using a base hash function secure hash algorithm SHA-1[13]. The advantage of using SHA hashing is that it randomly maps the hashed key values to a space of 160-bit binary numbers. The first *k*-1 bits (*k*=1,2..) will be extracted and a 1-bit prefix is added with bit '0' used for mice flows and bit '1' for elephant flows to form a *k*-bit index.
2. By using the index constructed as above, the pointer to the bucket is obtained from the index directory.
3. Lookup for the matched entry in the chosen bucket is carried out using the full 160-bit hashed value.
4. If there is no match in step-3, a new entry is made in the bucket, subject to space availability. If there is space constraint, a new bucket will be created using the bucket expansion mechanism as explained in the next section.

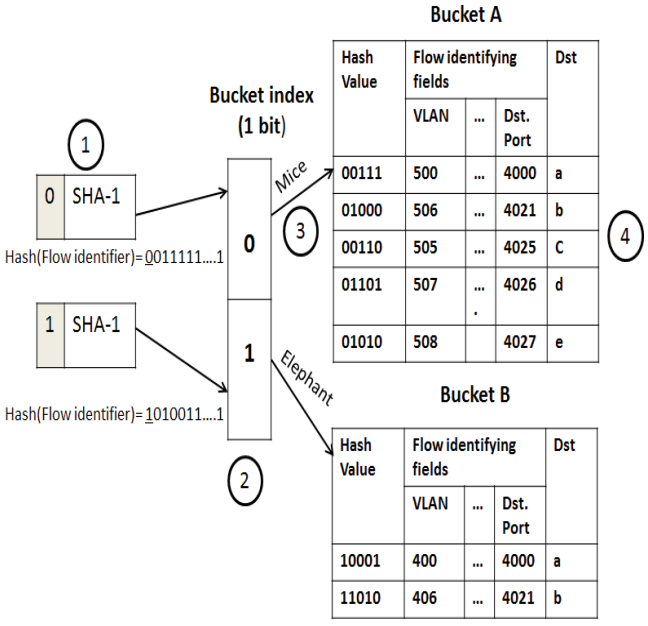

Figure 4. Illustration of buckets and Index tables

C. Overflow handling- Bucket Expansion

If a bucket overflows, the buckets size are doubled with the new index using one extra bit. For example, if an 1-bit index is used originally, after expansion 2-bit index will be used. As a result, the entries in an original bucket (say, with the index 0) will be distributed among two buckets (with index 00 and index 01). The index directory will also be doubled, i.e., if it originally contains 2^k index values, it will now contain 2^{k+1} entries, implying that the depth increases to k+1. We note that these k+1 bits are formed from the random hash value. As a result, it is likely that the new buckets are more balanced. If the 0-indexed mice bucket exceeds its capacity (1024 for example), it will be expanded into 2-bit indexed buckets: '00' and '01' thus increasing its maximum capacity to 2048. If there is a need for further expansion, it will be expanded to 3-bit indexed buckets: '000', '001', '010', and '011'. A similar procedure is used for elephant buckets with the prefix '1'.

We note that the number of index bits for mice flows and elephant flows need not be the same. Since the number of mice flows is much more when compared to the number of elephant flows, it is highly likely that the index size is larger for mice flows than elephant flows.

Bucket expansion example is illustrated in Figure 5. Here we assume a bucket size of 5. When the sixth mice flow arrives, bucket A overflows and a new bucket A' is used and the six entries (including the new one) are distributed over the buckets A and A' each of which now uses a 2-bit index.

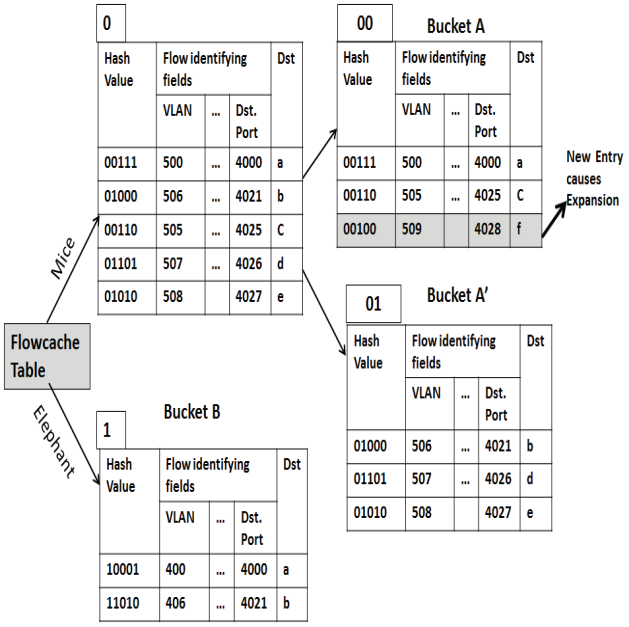

Figure 5. Illustration of Bucket Expansion

D. *Cache Replacement strategy*

Usually the cache size is small when compared to the total number of concurrent flows due to the memory cost and high sharing efficiency; and therefore replacement policy is needed. In our replacement strategy a new mice flow can evict only a mice flow and an elephant flow can evict only an elephant flow. We use localized least recently used (LRU) policy wherein, the least recently used flow in the local bucket is evicted. By doing so, we preserve the mapping of a flow to the bucket and ensure fast processing. The least recently used flow entry will be evicted in the event of a cache miss and permissible numbers of buckets have been used up. We note that although we use localized replacement, the hit ratio is likely to be high as evident from our performance study presented in Section IV.

E. *Bucket Shrinking*

Flows with long inactive period will be deleted from the cache and it might cause a bucket to become empty. In that case, a check is made for possible bucket shrinking wherein the bucket index shrinks by 1 bit and the number of buckets is halved and flow entries are shuffled accordingly. Since there are multiple Elephant flows in one bucket, it is unlikely for all of them to stay inactive for long period of time. Further, bucket shrinking occurs only if the number entries fall below the half of the bucket size which is less likely to occur frequently.

IV. Performance Evaluation

In this section, we present the results of our performance study carried out using software implementation of the cache. We compare the performance of our proposed method with the OpenFlow based wildcard-aware linear search table [6] as described in section II with different flow cache replacement policies such as Random and First-In-First-Out(FIFO). While Random method selects the flow to evict from the cache at random, FIFO evicts the flow which arrived first. Following the finding in [1], we generated the Data Center bursty traffic pattern which is a mix of 80% mice flows and 20% elephant flows using Ostinato Traffic generator [18].

A. *Performance in terms of Cache Hit Rate*

Figure 6 shows the cache hit rate achieved by the 4 cache replacement strategies: LRU, Wild-card linear, Random and FIFO with differential buckets for mice and elephant flows. We can observe that our proposed LRU method performs better than OpenFlow wildcard-aware linear table [6] replacement, Random replacement and FIFO replacement method. Also note that although the numbers of mice flows are large, the performance of elephant flows is not affected due to different buckets used for mice and elephant flows.

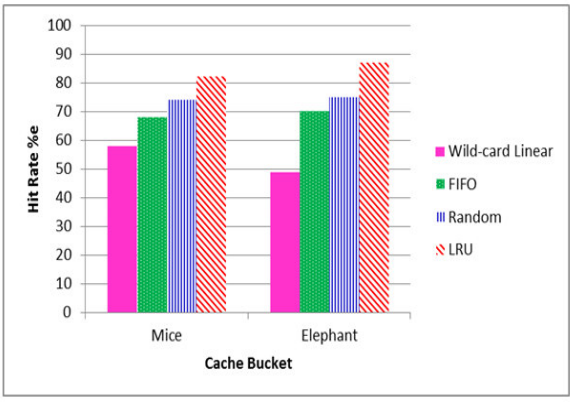

Figure 6. Comparison of cache Hit Rate

B. Impact of Cache bucket size

We next look at the impact of cache bucket size of 1k, 2k, 4k and 8k flow entries on the performance in terms of look-up time. Figure 7 shows the effect of the cache size on the look up time with wildcard-aware linear flow table and our proposed dynamic index hashing on the new architecture. We can observe that the lookup time increases with increasing bucket size. Further, by organizing the cache into different buckets, our method achieves significant performance improvement in terms of look up time when compared to wildcard-aware linear flow table. We set the cache bucket size as 8k which can store up to 8k flow states. Our experiment results suggest further increasing the bucket size will in turn increase the look up time beyond 3ms.

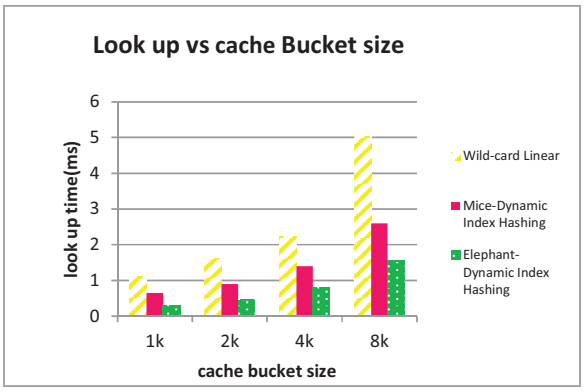

Figure 7. Impact of cache bucket size

C. Performance in terms of Lookup Time

We compare the performance of our method with wildcard-aware linear table in terms of lookup time. We measure the look up time based on simulation PC with custom implementation. The look up time includes the time taken to carry out hashing and the corresponding bucket look up time. Whereas in the Wild card method, the look up time refers the time for linear table – walk time. In Figures 8, 9 and 10, we show the CDF lookup time comparisons for 25th, 50th, 75th, and 100th percentile. From the graphs, we can observe that the lookup time by our method for both mice and elephant flows are considerably smaller than that by the wildcard-aware linear method. The performance improvement is more significant for higher percentiles (i.e. 75 and 100 percentile).

With the wildcard-aware linear method, there is no differentiation between the mice and elephant flows, and it is highly likely that the elephant flows are in the higher percentile region resulting in unfairness for elephant flows. This unfairness problem is reduced in our method. Also, we can observe that the lookup time for mice flows in higher than that for elephant flows. This is because there are more entries inside a mice bucket than an elephant bucket.

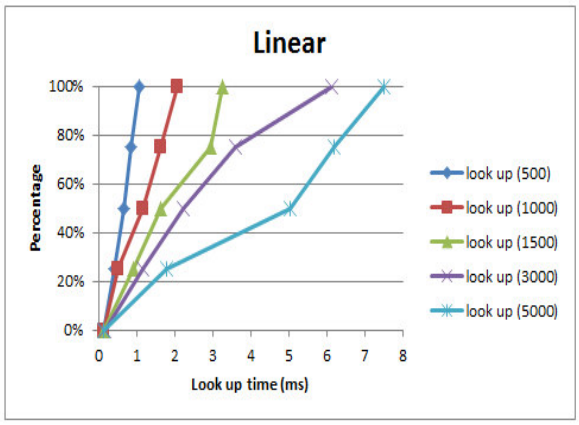

Figure 8 . Lookup Time for Wild-card linear

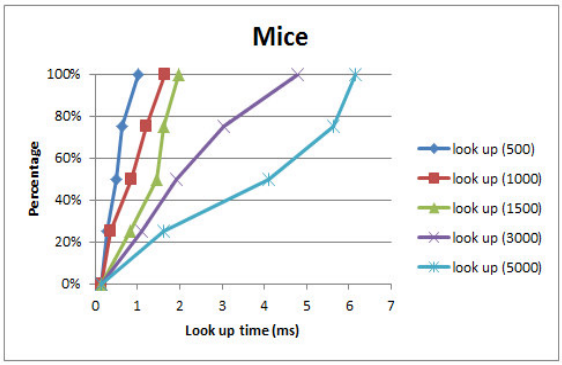

Figure 9. Lookup time for Mice

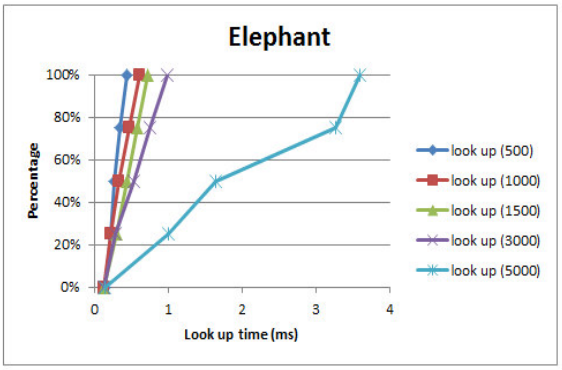

Figure 10 . Lookup time for Elephant

D. Bucket expansion overhead

In this section, we study the overhead due to bucket expansion. We assume the number of the flows in the bucket is 1000 and the mice and elephant flows are in the proportion 80:20. When the bucket expands, some of the entries in the original bucket will be moved to a new bucket incurring the overhead whose values are plotted for three different values of the number of flows in Figure 11.

We note that when the number of flows exceeds 1024, we need a 2-bit index and when it exceeds 2048 we need a 3-bit index. As such, for the case of 5000 entries (total), the mice flow buckets need to be expanded from 1-bit to 2-bit index then to 3-bit index; and for elephant flow buckets to expand from 1-bit to 2-bit index. For the case of 3000 entries (total) the mice flow bucket needs to expand from 1-bit to 2-bit index and then to 3-bit index. For the case of 1500 entries (total) the mice flow bucket needs to expand from 1-bit to 2-bit index. We can observe that the overhead is small but it increases with the increasing number of index bits. At the same time, we note that such an expansion occurs less frequently compared to the lookup operation.

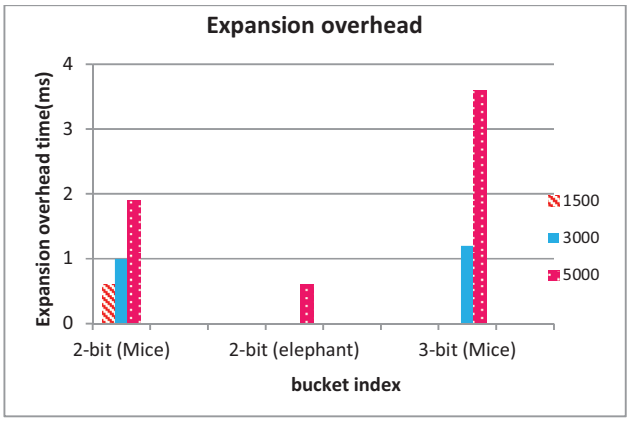

Figure 11. Bucket Expansion Overhead

V. CONCLUSION

In this paper, we addressed the problem of improving fairness for elephant flows in a data center as they suffer from frequent eviction because of the flooding of mice flows. We adopted a differentiated approach and proposed flow cache architecture with dynamic-index hashing for placing the flow records onto various buckets. We also proposed a localized LRU-based replacement strategy, which gives the best performance Overall the differential flow cache architecture provides a simple and effective means to address the overload on the controller due to flow table exhaustion at the OpenFlow switch.

REFERENCES

[1] T. Benson, A. Akella, D.A. Maltz, "Network traffic characteristics of data centers in the wild", in: Proc. ACM IMC (Melbourne, 2010), pp. 267–280.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L.Peterson,J. Rexford, S. Shenker, and J. Turner. ,"OpenFlow: enabling innovation in campus networks", SIGCOMM Comput. Commun. Rev., 38(2):69–74, 2008.

[3] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. Devoflow: Cost-effective flow management for high performance enterprise networks. In ACM SIGCOMM Hot-Nets Workshop, Monterey, CA.

[4] Mahadevan, P., Sharma, P., Banerjee, S., Ranganathan, P.: A power benchmarking framework for network devices. In: Proceedings of the 8th International IFIP-TC 6 Networking Conference. NETWORKING '09 (2009)

[5] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying NOX to Data center", In Proc. of 8th ACM Workshop on Hot Topics in Networks, 2009.

[6] OpenFlow Reference System, http://www.openflowswitch.org/

[7] Martin Zadnik, Marco Canini. "Evolution of cache replacement policies to track heavy-hitter flows," in Proc. of PAM'11, 2011.

[8] T.Pan,X.Guo,C.Zhang,W.Meng,B.Liu, " AlTF: A Replacement Policy to cache Elephant Flows in th Presence of Mice Flooding", in Proc. of ICC'12.

[9] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," In Proc. 22th Ann. IEEE Infocom (Infocom 03), IEEE Press, 2003, pp. 42-52.

[10] V.C. Ravikumar and R.N. Mahapatra, "TCAM Architecture for IP Lookup Using Prefixes Properties," IEEE Micro, vol. 24, no. 2, Mar.- Apr. 2004, pp. 60-69.

[11] w.wu,D.zi,y.Lan,T.wu,"Power aware TCAMS for routing table Look up ", In Proc. of IEEE GreenCom 2010.

[12] FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, R. , "Extendible hashing: A fast access method for dynamic files.", ACM Trans. Database Syst. 1979.

[13] M. Yu, J. Rexford, M. J. Freedman, and J. Wang.," Scalable flow-based networking with DIFANE", In Proc. of ACM SIGCOMM 2010.

[14] A.R.Curtis, "Mahout:Low overhead datacenter traffic Management using end-host based elephant detection," In. Proc. of IEEE INFOCOMM 2011

[15] FIPS 180-1 *Secure Hash Standard, Apr 1995*.

[16] OpenFlow Sswitch Specification –Ver.1.3.0, The Open Networking Foundation, 2012.

[17] Renuga Kanagavelu, Luke Ng Mingjie, Khin Mi Mi, Francis Bu-Sung Lee, Heryandi,"OpenFlow based control for re-routing with differentiated flows in Data Center Networks", ICON 2012.

[18] "Ostinato – Packet/traffic Genarator and Analyzer", http://code.google.com/p/ostinato/