Flowcache: A Cache-Based Approach for Improving SDN Scalability

Atin Ruia, Jasson Casey, Sujoy Saha, and Alex Sprintson
Texas A&M University, College Station, TX
{atinruia, jasson.casey, sujoy_saha, spalex}@tamu.edu

***Abstract*—Today, the transition of traditional networking model to SDN-type architectures poses several major challenges. In typical SDN settings, the routers and switches frequently generate requests to the controller which ensures the proper and efficient operation of the network. Increased workload on the controller results in larger control plane response times, which, in turn, leads to a delayed response to the data plane events. As a result, the controller can become a major performance bottleneck which will negatively affect the performance of the overall system. This problem is important in access and edge networks where a controller is expected to serve a large number of remote switches.**

In this paper, we propose to address this problem by applying the principle of caching to the control plane of the SDN framework. In particular, we propose to augment an SDN architecture with a transparent layer in between the controller and switch, referred to as a *flowcache*. Flowcache acts as a software cache for the control traffic, by temporarily storing the content of recent flow table entries modified by the switch. This results in a significant reduction of the access time for future requests of similar flows. We analyze different design choices for the flowcache, analyze its properties, and evaluate the benefits of introducing a flowcache in an SDN architecture.

I. Introduction

Software Defined Networking (SDN) has recently gained popularity within the networking industry. The SDN model provides flexibility in both designing and managing communication networks with a logically centralized controller. In SDN architectures, the controller maintains a centralized view of the underlying network and installs flows in the forwarding elements (switches) to route traffic along specific paths. This approach allows network administrators to configure networking devices to meet the requirements of the network applications.

SDN applications have requirements on the minimum size of the flow tables at each switch for their proper operation. For a large number of applications this count grows multifold. Since each hardware switch has limited capacity, it can store only a small number of flows. This forces the controller to reinstall flows on a frequent basis, increasing the packet processing latency by the data plane. Therefore, an increase in the number of applications running on the controller can lead to a substantial performance degradation of the entire system.

Switching functions can be implemented in software or hardware. Software switches, such as Open vSwitch [8], are not constrained by the flow table capacity and have unbounded table size, but are limited by the number of ports and slow processing of packets. Hardware switches implement packet processing pipelines leading to significant performance improvements. They typically implement flow tables using Ternary Content Addressable Memory (TCAM) for faster classification of packets. Today's commodity switches can support up to 20,000 flow rules. However, supporting a large number of flows results in an excessive overheads in terms of costs and power requirements.

In this paper, we leverage the principle of caching to minimize delays and maximize throughput of SDN-enabled hardware switches. Our architecture is primarily targeted at the edge and access networks. Access networks interconnect the end-users to the core network using wireless and/or wired connection interfaces. These networks typically employ inexpensive commodity switches with limited flow tables capability and have low bandwidth management channel to the controller. Our goal is to leverage caching principles to maximize performance in these settings.

In our architecture, a *flowcache* sits transparently between the controller and the switches. The *flowcache* acts as a software cache for the logically centralized controller, by storing all the flows recently installed into the switch. Since flowcache is a software based cache, it can store a large number of flows. This gives the controller an abstraction of a switch with a large amount of flow table space, thus satisfying the table space requirements of all the applications concurrently running on the controller. In this paper, we have the following goals:

- Analyze different design choices and tradeoffs of adding a *flowcache*;
- Evaluate the performance improvement by inserting the same in an existing SDN architecture in terms of throughput and latency values.

Related work. Several studies have focused on the scalability problem in SDN architectures. These studies have identified three separate bottlenecks in an SDN model - the controller, the communication channel across the control plane and data plane, and the hardware switch.

In Kandoo [3], the authors propose a hierarchical controller design, where the local applications are offloaded to the local controllers, while the applications requiring global view of the network, execute on a centralized one. This design requires maintenance of complex data structures between the global and local controllers. Onix [6] and Hyperflow [12] present a distributed controller design, where each controller maintains a shared global network view. Industry supported distributed controllers such as ONOS [1] and Open Daylight [7] are designed with high scalability and availability in mind. However, similar to other distributed controllers they provide a weak consistency model. In addition, the code complexity of such distributed controllers may lead to system-wide instability [4].

Difane [14] reduces traffic in the controller-switch channel by partitioning the flows among switches and installing appropriate rules to selectively direct packets to specific switches. Katta et al. [5] presented a new hardware-software architecture called CacheFlow. CacheFlow selectively redirects

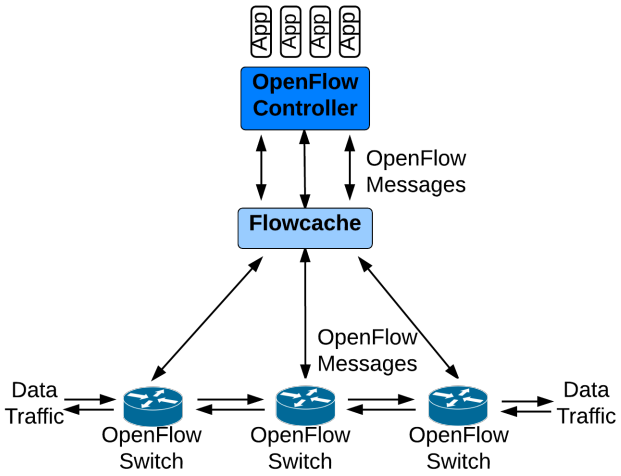


Fig. 1: SDN Architecture with *Flowcache*

data packets to software switches (acting as secondary cache) on a hardware switch miss. This solution, however, targets the data center scenarios and requires additional data path resources, in particular, additional software switches and high bandwidth communication links between the hardware and software switches. In contrast, the Flowcache solution does not require any additional data plane resources, hence it can be implemented at low cost with little additional overhead. This makes our solution more appealing for the edge/access network scenarios.

II. FlowCache Design

The proposed SDN architecture with the flowcache component is depicted in Figure 1. The *flowcache* is located between the controller and the switches and is transparent to every other component. *Flowcache* acts as a middle box, intercepting the OpenFlow (OF) messages being sent across the management links (i.e., the links between the controller and the switches) to monitor the required behavior of the switch.

The following sections provide an overview of the different properties of *flowcache*, the possible locations it can be deployed, and the way it handles the OF messages sent across the management link.

A. Flowcache Properties

Flowcache allocates a flow table for each table present in the switch. Since these tables are maintained in software, their capacity is "unbounded," i.e., there is practically no limit on the number of flows stored in each table due to the low cost of system memory. Each table constantly maintains a superset of all the flows installed in the corresponding flow table in the switch. In case of a PacketIn event (a message sent from the switch to the controller that contains a captured packet), *flowcache* searches its own flow tables for a header match. On a hit, it sends a FlowMod/PacketOut to the switch. In case of a miss, it sends a PacketIn message to the Controller.

Flowcache accepts connections from the switch acting as a controller, and connects to an actual controller representing itself as a switch. Neither the controller, nor the switch is aware of the presence of the *flowcache* component.

B. Flowcache Location

Flowcache can be deployed at three possible locations in the SDN architecture:

- **In the controller** - Located beside the controller the *flowcache* shares its burden by managing all the switches trying to connect to the controller. In this model, all incoming packets initially traverse through flowcache before they hit the controller. However the latency across the *flowcache*-switch channel increases, leading to higher communication delay across the management link.
- **As a middle box** - In this scenario, a logically centralized controller manages switches present in multiple wide area networks (WANs). A flowcache installed in each of these WANs provides faster classification of flows and reduces the communication delay across the management links. It reduces the load on the controller by caching all similar types of flows present in a WAN like ISP flows etc.
- **In close proximity to the hardware switch** - In this scenario, the *flowcache* manages a small number of switches (in same LAN) with faster processing of flows in the flowcache flow table. Further, the communication delay across the flowcache-switch channel decreases leading to better throughput of the overall system. In this model, a larger number of *flowcache* devices need to be deployed compared to the previous scenario.

In this paper, we focus on the scenario in which the *flowcache* is located at close proximity to the OF enabled switches to minimize the communication delay across the flowcache-switch channel. The *flowcache* is designed with a primary focus on access networks which have few commodity switches connected to a master station. The master station then connects to the controller. In our design, the *flowcache* will be located in the master station.

C. Handling Openflow Messages

The flowcache transparently interprets all the OF messages. The two important categories are:

1. **Modification/Update messages:** The OF protocol defines FlowMod message to be used to add, modify or delete flows from a specific table in the switch. On receiving a FlowMod message, the flowcache initially updates its corresponding flow table, then updates flow table in the switch (Fig 2). The flowcache handles the overflow error condition, by evicting a flow based on the eviction parameter. The other modification messages are simply redirected to the switch.
2. **Traffic Statistics:** *Flowcache* polls the switch for statistics of all the installed flows at fixed intervals of time. On receiving a Multipart StatsReq message, it queries the switch with the request. However, for a StatsReq of an inactive flow present in the flowcache, it simply responds with the statistics information maintained in its own flow table.

III. Eviction Policies

Flowcache acts as an *inclusive* cache. It buffers all the flows being installed by the controller to the hardware switch. For each such flow, it stores metadata, which helps to maintain consistency between the switch and the cache. A flow is classified as active when installed in the switch; otherwise, it is classified as inactive. Periodically, all the active flows are updated using the flow statistics obtained from the switch.

Flowcache evicts a flow from the switch when a switch table is completely filled or has reached a certain threshold.

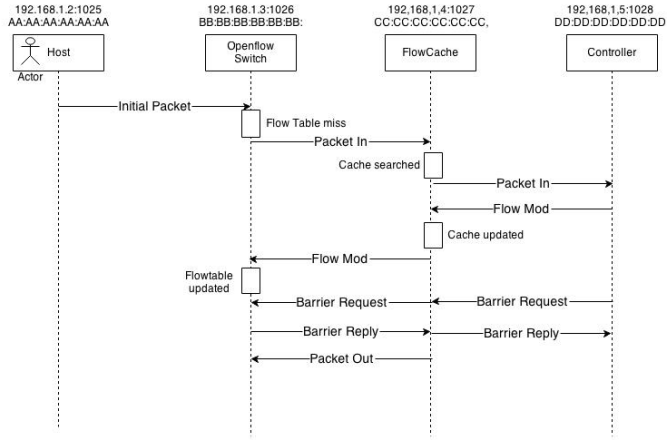


Fig. 2: Flow Modification Scenario

Different eviction strategies can be deployed as discussed in Section III-C. However, in each eviction strategy, flowcache needs to consider the different types of dependencies that exists between flows. There are two main types of dependencies that needs to be considered:

A. Intra-table Dependencies

A flow table consists of a fixed number of flows sorted on priority and exact match rules. The controller assigns priority to flow rules, which along with the match structure decides the packet routing. By adding a flowcache in the SDN architecture, we may need to periodically evict flows from a table. However, we must ensure that all packets are processed using the same rules, irrespective of the presence of flowcache.

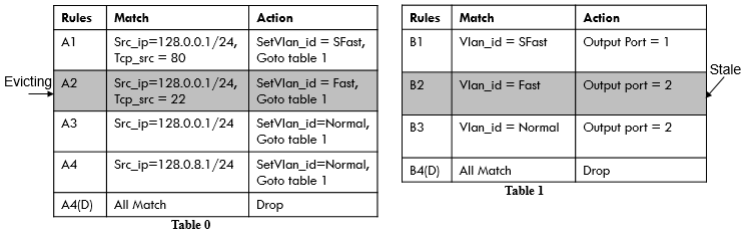
To solve the above problem, we use the heuristic presented in [5] that extracts the flows from the switches in dependent sets. In our implementation, our goal was to strike an approximate balance between the minimum number of flows to be evicted, and the minimum amount of traffic traversing these flows.

B. Inter-table Dependencies

Starting with OF version 1.1, a switch allows the controller to have fine-grained control over multiple flow tables. Multiple flow tables simplify flow management and reduce explosion in the number of flow entries.

In OF protocol, a packet traversing the switch dataplane initially hits the first table. It advances from table i to table j, where $0 \leq i < j \leq n$, on executing an instruction of type GOTO Table. Thus, all packets hitting table j are redirected from a prior table i.

On eviction of a flow entry with an instruction of type GOTO Table 'T', an entry in the corresponding flow table 'T' can become stale. To illustrate the above scenario, assume two OF tables in a switch say Table 0 and Table 1 (Figure 3). Table 0 acts as a security firewall table, where it rejects all flows not originating from IP addresses 128.0.0.1/24 and 128.0.8.1/24 and classifies packets based on its source of origin. It can direct packets through different output ports based on the type of service expected. By Table 1, all data packets with VLAN id "Fast" are sent through the output port 2. Now, say we decide to evict the flow entry A2. Since A2 is the only entry which sets the packet VLAN id to "Fast", evicting it would



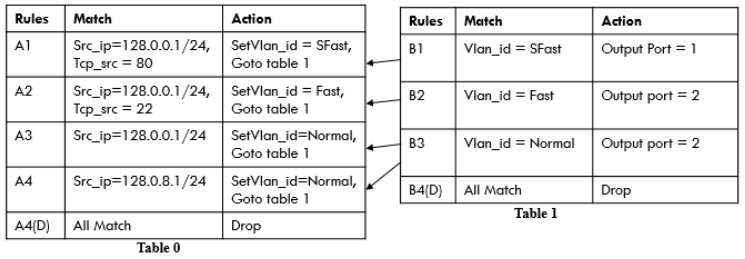
Rules	Match	Action
A1	Src_ip=128.0.0.1/24, Tcp_src = 80	SetVlan_id = SFast, Goto table 1
A2	Src_ip=128.0.0.1/24, Tcp_src = 22	SetVlan_id = Fast, Goto table 1
A3	Src_ip=128.0.0.1/24	SetVlan_id=Normal, Goto table 1
A4	Src_ip=128.0.8.1/24	SetVlan_id=Normal, Goto table 1
A4(D)	All Match	Drop

Table 0

Rules	Match	Action
B1	Vlan_id = SFast	Output Port = 1
B2	Vlan_id = Fast	Output port = 2
B3	Vlan_id = Normal	Output port = 2
B4(D)	All Match	Drop

Table 1

Fig. 3: Table Highlighting Stale Entries



Rules	Match	Action
A1	Src_ip=128.0.0.1/24, Tcp_src = 80	SetVlan_id = SFast, Goto table 1
A2	Src_ip=128.0.0.1/24, Tcp_src = 22	SetVlan_id = Fast, Goto table 1
A3	Src_ip=128.0.0.1/24	SetVlan_id=Normal, Goto table 1
A4	Src_ip=128.0.8.1/24	SetVlan_id=Normal, Goto table 1
A4(D)	All Match	Drop

Table 0

Rules	Match	Action
B1	Vlan_id = SFast	Output Port = 1
B2	Vlan_id = Fast	Output port = 2
B3	Vlan_id = Normal	Output port = 2
B4(D)	All Match	Drop

Table 1

Fig. 4: Reference Edges Between Flow Entries

imply none of the packets match the flow entry B2 in Table 1, i.e. the entry B2 becomes stale.

A possible solution to the above problem is to identify and evict stale entries during eviction of normal entries. To identify stale entries, we construct a reference graph. In this graph, a node represents a flow entry, whereas an edge represents a reference between two flow entries. Whenever an entry 'E' containing an instruction of type GOTO Table 'T' is inserted, we formulate the packet set 'S' that matches the newly inserted entry, and advances to the next table 'T'. For all rules in table 'T', we find the intersection of the packet set 'S' and their match set. On every successful match, we add a reference edge between 'E' and the matched rule.

Consider another scenario. On insertion of a simple flow rule, we need to verify if a reference edge needs to be added between a prior table rule and new rule. For that purpose, each table keeps track of all the entries, where it has been referenced. In a similar way, we would try to find the intersection of the set of packets originating from these entries, and the newly added entry. On success, we would add a reference edge from the earlier entry to the new entry. Figure 4 shows the reference edges being added between flow entries.

On eviction of a flow, we would validate the state of all the reference edge nodes. For all the nodes prior to the current table, we would remove the edge. For all other nodes, we would verify the incoming edge count. A NIL count indicates the node has become stale and has to be evicted out.

The above solution requires an exhaustive search of flow entries at the time of insertion of a rule. Further, the packet set S matching a flow entry can have a wide range of values, given the large number of fields packets can be matched on.

In our implementation, we employ an easy and efficient solution. We avoid evicting stale entries at the time of removal of an entry, and depend on our eviction policy to handle these entries. Since the stale entries do not match on a data packet, the eviction strategy should remove these entries first. In this way, we deal with only current entries and reduce the total time taken during eviction.

C. Eviction Strategies

Hardware switches perform better in comparison to software switches. However, these switches are limited by hardware resources such as TCAMs, queues, etc. Managing these resources efficiently becomes necessary for improved network performance.

Eviction in a flow table is a known problem. Inherently, in OF protocol, the controller can evict a flow from the switch in two ways (i) Request of the Controller - FlowMod type DELETE; (ii) Switch Flow Expiry Mechanism - Idle or Hard timeout value.

However, selecting the flow to be evicted is a challenging task. Different studies have tried to address this problem based upon the application requirements and flow statistics. Broadly, eviction in a switch can be classified depending upon the locality of flows.

- Spatial Locality - In this type of locality, packets hitting a particular flow in the switch provides information of future packets. Here, the locality is often dependent upon the type of application running on the controller. For example, say we have a DHCP application currently running on the controller. When the controller application receives a PacketIn message with a DHCP DISCOVER packet, it can pre-install flows on the switch based on the future expected DHCP messages like DHCP OFFER and DHCP REQUEST messages.
- Temporal Locality - This locality is based on the network load. Higher number of packets hitting a flow indicates a higher importance to that flow, while a flow with low packet hit in the recent past indicates a less important flow. This type of locality tries to extract metadata information from the frequency of data packets, rather than the type of packet. Common eviction strategies such as LRU, FIFO are based on the temporal locality.

A *flowcache* acts as a transparent component managing the table space in a switch. It is unaware of the type of applications currently running on the controller. Thus, for eviction it relies on the temporal locality of the flows and upon the packet statistics.

In the next section, we discuss three different type of eviction strategies that can be used by *flowcache*.

1) Eviction done by Switch: When a switch table is completely full, or a maximum threshold has been reached, the switch evicts a flow based on the type of eviction policy currently in use. Starting OF 1.4, the OF protocol allows the controller to configure the switch to evict flows either based on the importance of a flow, or based upon lifetime of a flow.

Although the above strategy is well defined, none of the switches except OpenvSwitch supports OF protocol 1.4. Given the complexity of the OF protocol and its frequent iterations, it is difficult for vendors and open-source organizations to maintain the most recent OF standard. Further, the current message structure, does not provide dependency management.

2) Eviction using Flow Statistics: In this strategy, the flowcache evicts flows from the switch, using the measure of flow statistics. OF 1.3 protocol defines messages Multipart-Stats Request/Reply which help the controller to obtain flow statistics from the switch.

Flowcache internally maintains packet count for all flows. Periodically, it updates the count by requesting the switch for flow statistics of all the flows currently residing in the switch.

Flowcache can maintain either coarse-grained or fine-grained statistics for a flow depending upon the periodicity of the request. The periodicity itself depends upon the available bandwidth across the flowcache and switch channel. Although fine-grained statistics provide a near real-time view of specified switch table , it overloads the switch with frequent requests. Since a commodity switch can contain $\sim$ 2K flows, frequently obtaining the statistics of all the flows will put the available bandwidth across the control channel under pressure.

3) Eviction using Timeouts: In this strategy, flowcache tries to setup flows with near perfect idle and hard timeout values. Recent studies by Zarek et al. and Vishnoi et al. ([15], [13]) have worked on different algorithms to predict perfect idle time timeout values.

A small timeout value leads to early eviction of a flow from the switch, increasing the PacketIn count to the controller. However, a large timeout value leads to flows being resident on the switch for a longer duration. This in turn leads to an increase in the *'working set of flows'* - the number of flows residing in the switch at the same time, thus requiring a larger switch table size. Therefore, it is critical to setup flows with near perfect timeout values. The works [15], [13] present heuristic based timeout values.

IV. Experimental Setup

We constructed an experimental model to replicate the scenario of an access network. Access networks typically consists of wireless/wired access points communicating to a nearby master station, which in turn communicates with the remote controller.

Our model in Figure 5 consisted of the Ryu [9] controller, the Userspace Softswitch [10], two Linux hosts and a *flowcache* component. The controller was installed on a remote machine resident on one of the GENI [2] servers located in the Texas A&M Exo GENI rack. The switch and the hosts resided on a Linux machine, while the *flowcache* operated from a separate Linux machine connected to the same LAN. The two hosts connected to the switch via Virtual Ethernet pairs to form a simple linear topology. The latency across the *flowcache*-controller channel was found to be around 4ms. The setup was based on latencies found in access networks.

Different experiments were conducted to compare the performance of the SDN model with and without *flowcache*. In order to test the efficiency of the system, we measured the available throughput across the two hosts. The load on the controller was measured by calculating the number of PacketIn OF messages received by the controller. Since the controller has to process each of the PacketIn message, higher number of PacketIn messages increases the load on the controller.

The workload traffic was generated using features from CAIDA [11] traces for normal Internet traffic. The packets were captured on a 10GB line card and had an average throughput of around 3GBps in 60 seconds window. Due to privacy reasons, the traces were anonymized, and data link-layer headers and the packet data portion were deleted.

In our experiment, we scaled down the packet trace to run with an average throughput of around 22Mbps. Since, the trace provided only layer3 and layer4 headers, we had to create our own network traffic. We installed a UDP client and a packet sniffer at two end hosts. The UDP client would send packets to the packet sniffer ip, changing the UDP destination ports to the layer 4 ports obtained from CAIDA packet trace. Using

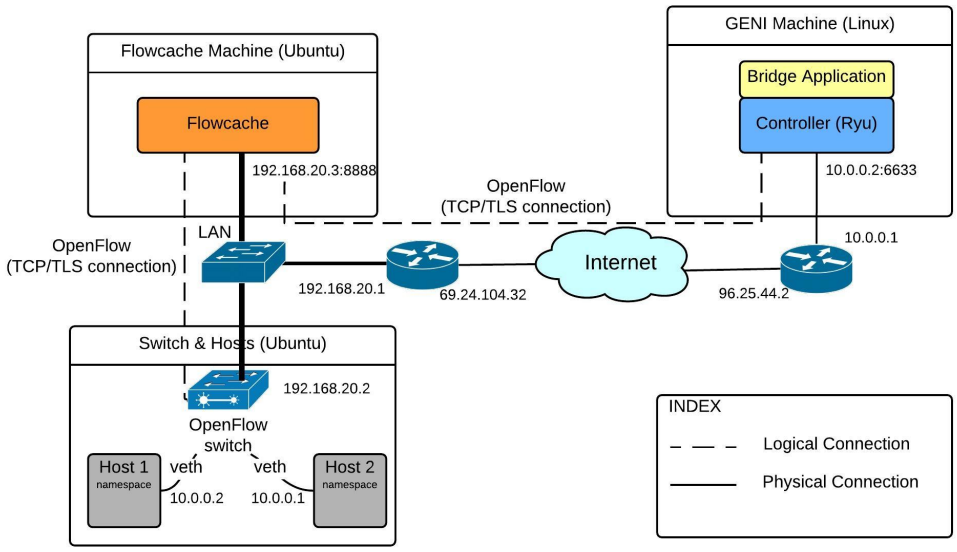


Fig. 5: Experimental Setup

this technique we generated individual flows. In our test, we inserted around 200,000 packets in the network at an average rate of 22Mbps. These 200,000 packets resulted in generation of around 24,500 individual flows.

V. Results and Discussion

We evaluated the performance of our system by measuring the throughput across the end hosts. We examined the change in throughput by varying the size of the switch flow table. In our model in Figure 5, the latency across the *flowcache* - controller channel and switch-controller channel was configured as 4 ms, while the latency across switch-flowcache channel was around 0.5 ms. In Figure 6, the observed throughput for *flowcache* remained constant for all the table sizes at the set input rate. However, the performance of the base SDN model (i.e. without *flowcache*) constantly decreased with smaller table sizes. In this experiment, *flowcache* sent around 26000 flow requests to the controller compared to around 100000 flow requests sent in the base case. The high count of flow requests in the 4 ms latency channel caused the performance decrease in the base SDN model.

Figure 7 shows the difference in the number of messages processed by the controller. An increase in the size of the flow table led to higher hit rate, leading to less number of PacketIn messages being sent to the controller. The SDN-*flowcache* component sends PacketIn message only on finding a new flow. Since *flowcache* buffered all the flows installed by the controller, it avoided sending a request for the same packet it had encountered before. This led to smaller number of PacketIn messages to the controller, producing an almost constant load across all table sizes. The number of PacketIn messages sent across the *flowcache*-controller channel was considerably less when compared to the base SDN model, thus showing a significantly reduced controller load.

Figure 8 shows the average time needed by a packet (in micro seconds) to get serviced by switch. Notice that after the initial flow rules are stored into *flowcache* there is a massive drop in average processing time for a packet. This

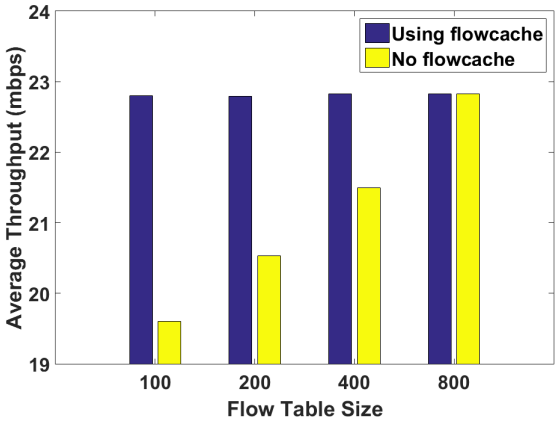


Fig. 6: UDP Throughput Comparison for Different Table Sizes

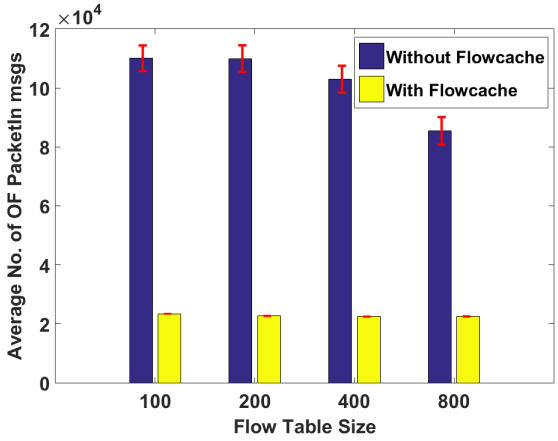


Fig. 7: Load on Controller for Varying Flow Table Size

improvement in performance is due to fast access of flow-

rules set in *flowcache* when a table miss happens instead of sending to controller.

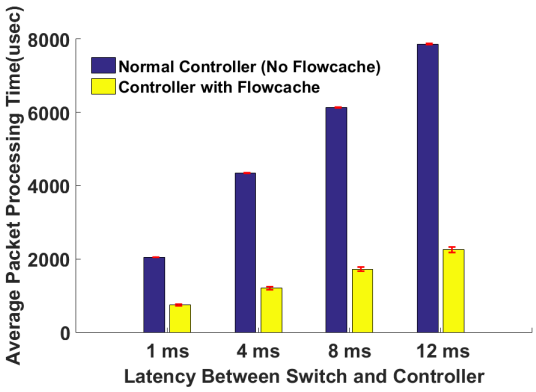


Fig. 8: Access Time(usec) vs. Packet Count

Next, we contrast the performance obtained by using statistics measurements based eviction strategies used by *flowcache* at different periodicities. Coarse-grained statistics were obtained at an interval of 10 seconds, whereas fine-grained statistics were obtained at a small interval of 1 second. Fine-grained statistics presents a near real-time view of the switch, but puts additional pressure on the switch and the communication channel. In our experiment, the fine-grained statistics showed better performance for tables sized 400 and 200 (Figure 9). In these two cases, the number of PacketIn messages were comparatively less, and evicting flows based on real time view of the tables resulted in better performance. However, for table sized 100, the state of the table changed rapidly, so in fine-grained eviction policy the additional flow statistics failed to provide real-time view of the tables and only added excess traffic to the control plane channel.

VI. Implementation Details

Flowcache is currently developed for OF 1.3 protocol by modifying the Softswitch [10] codebase. *Flowcache* acts as a middle box which accepts connections from the switch as a controller. For each initiated connection request, it opens a new connection to the controller. Being transparent, it exploits the use of VLAN tags to direct packets to the controller in case of a table-miss entry. *Flowcache* handles soft or hard timeouts

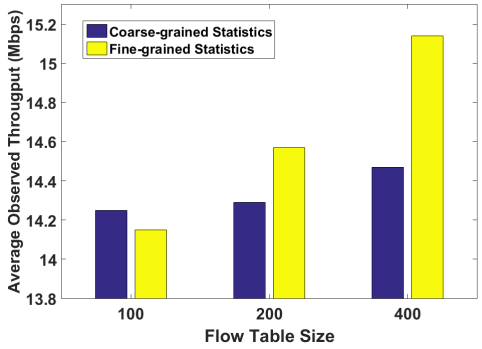


Fig. 9: Throughput - Coarse-grained vs. Fine-grained Statistics

by requesting a Flow Removed message in the event a flow is removed from the switch. This helps flowcache to keep track of all the active flows in the switch.

VII. Future Work

Flowcache can be extended in different directions. It can be extended to solve the compatibility issues between different OF versions. Currently, a number of OF switches support separate versions of the OF protocol. In this situation, it becomes difficult for a controller to manage these switches. An application developer is either limited by the base set of capabilities supported by all the switches, or feels the need to manage the capabilities of each switch separately. In such cases, *flowcache* can present 'a big switch' abstraction to the controller, where it provides the controller application an interface to the advanced set of switch capabilities. All capabilities not handled by a switch are handled internally by *flowcache* using a software switch.

Flowcache can also extend its support for all the dataplane abstractions. Currently, *flowcache* only supports the switch flow table abstraction. New abstractions like groups, queues and meters can be individually handled by *flowcache*.

References

[1] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[2] Geni - Global Environment for Network Innovations. Retrieved June 11, 2014 from https://www.geni.net.

[3] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[4] Jin Won Kang, Sae Hyong Park, and Jaeho You. Mynah: Enabling lightweight data plane authentication for sdn controllers. In *Computer Communication and Networks (ICCCN), 2015 24th International Conference on*, pages 1–6. IEEE, 2015.

[5] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 175–180, New York, NY, USA, 2014. ACM.

[6] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[7] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[8] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[9] Ryu - SDN Framework. Retrieved March 20, 2015 from http://osrg.github.io/ryu.

[10] Soft Switch. Retrieved March 20, 2015 from https://github.com/CPqD/ofsoftswitch13.

[11] The Caida UCSD anonymized internet traces 2014. Retrieved March 20, 2015 http://www.caida.org/data/passive/passive_2014_dataset.xml/.

[12] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX Association, 2010.

[13] Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. Effective switch memory management in openflow networks. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 177–188, New York, NY, USA, 2014. ACM.

[14] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with difane. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM.

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