An Effective Swapping Mechanism to Overcome the Memory Limitation of SDN Devices

Antonio Marsico, Roberto Doriguzzi-Corin and Domenico Siracusa
CREATE-NET - Fondazione Bruno Kessler, Trento, Italy
Email: {amarsico, rdoriguzzi, dsiracusa}@fbk.eu

***Abstract*—Thanks to its 1-cycle lookup performance, the Ternary Content Addressable Memory (TCAM) is considered an essential hardware component for the deployment of high-performance Software-Defined Networks (SDN). Unfortunately, in many network scenarios, TCAMs can quickly fill due to their limited memory size, thus preventing the installation of new flow-rules and leading to inefficient traffic forwarding. This issue has already been addressed in computer programming, where *Virtual Memory* is offered to applications to mimic a much larger physical memory, by swapping memory pages to disk.**

In a previous work, we proposed and discussed the architecture of a Memory Management System (MMS) for SDN controllers that, like the analogous process for computer Operating Systems, optimizes the memory usage and prevents anomalies due to lack of memory space. This work proposes a memory swapping mechanism for SDN controllers, a function of the MMS which gives SDN applications the illusion of unlimited memory space in the forwarding devices, without requiring any hardware modification or changes in the control protocol. The paper discusses the memory swapping mechanism design, its implementation and proves its quality using real traffic traces, demonstrating lower TCAM memory utilization and potentially increased network performance in terms of end-to-end throughput. A prototype of the MMS is available for testing as an open source project.

***Index Terms*—Software-Defined Networking; TCAM; ONOS.**

I. Introduction

In Software-Defined Networking (SDN), network applications rely on SDN controllers to handle the incoming traffic, via instructions that are executed by network devices. This piece of information is commonly saved in the memory of the devices and it is called *flow entry*. In the OpenFlow protocol, a widely used standard for SDN [1], flow entries can be of variable length/size according to the specificity of the action that is requested to the networking device (e.g. L3 routing, L4 firewalling, etc.). With respect to traditional L2 and L3 forwarding, OpenFlow allows a much finer control of the network traffic, but, at the same time, it requires more memory space for each flow entry. For instance, recent versions of OpenFlow require up to 773 bits for each entry[1], while 60 bits are sufficient to identify a flow for L2 forwarding (destination MAC address plus VLAN identifier).

Modern network switches are usually equipped with two different types of memory in which forwarding instructions are stored: *Binary* and *Ternary* Content Addressable Memory (BCAM and TCAM). Both memory types can do lookups in one clock cycle and in parallel fashion, therefore they are very efficient when matching the incoming traffic with the forwarding rules they store. However, BCAMs only provide binary lookups, so they return either 0 or 1. They are most useful for building tables that search on exact-matches such as MAC address tables. On the other hand, TCAMs can store three bit states (0, 1, and "don't care") and are the most commonly used mainly because they provide line-rate lookups. TCAMs work very well in conjunction with OpenFlow, where the flow table entries foresee a wildcard bit, used to inform the switch to ignore the value of the specified header field. Unfortunately, TCAMs are very expensive, power hungry and have a considerable footprint size with respect to the number of supported flow entries. Therefore, vendors tend to install TCAMs with very limited capacity, which can quickly get full, leading to inefficient forwarding operations.

This problem is partially mitigated in the switches' firmware, which combine BCAMs (SRAM, DRAM, etc.) and TCAM to build the flow table: BCAMs for exact-match entries, the TCAM for wildcard entries. However, this strategy is not suitable for SDN-enabled switches, where exact-match rules are rarely used.

Starting from specification 1.4.0, OpenFlow introduces two mechanisms to allow the SDN developer to handle the lack of memory space available on network devices: *eviction* and *vacancy events*. The first one enables the switch to automatically delete the flow entries with lower importance. The degree of importance of each single flow entry is set by the SDN applications. The second mechanism enables the controller to get an early warning based on a capacity threshold set by the SDN application. However, such approaches force the SDN developers (and the SDN applications) to take care of the memory utilization.

In literature, several works have been proposed to tackle the problem of the limited TCAM memory space, all with very different approaches. However, as reported in Section V, they impose significant constraints on the network architecture, changes of the switches' software or hardware, modifications of the control protocol, or the usage of exact-match rules in BCAMs to save TCAM space.

In our work, we advocate that SDN controllers shall grant their applications a reliable access to network devices' memory. To this purpose, we propose a Memory Management System (MMS) for SDN that aims at improving the TCAM usage and, consequently, the robustness of the network. Compared

[1]OpenFlow Switch specification v1.5.1 [2], including optional fields.

978-3-901882-89-0 @2017 IFIP

to the existing approaches, the MMS transparently optimizes the usage of the available memory by automatically removing the flow entries installed by applications that are no longer running (we called this feature *memory de-allocation*) and automatically moving the least used wildcard entries to a slower memory outside the switches when the TCAM is full (*memory swapping*).

We presented requisites and architecture of the MMS in [3], while we demonstrated our prototype of the *memory de-allocation* in [4]. This paper focuses on the design, implementation and evaluation of the *memory swapping* mechanism.

The term *memory swapping* recalls a technique used in computer Operating Systems (OSs) to exchange *memory pages* between the fast but limited in size Random Access Memory (RAM) and the slow but much larger hard disk. Especially in the past, when the RAM capacity was often not sufficient for multi-tasking environments, *memory swapping* was the only way to make the applications seamlessly run even in low memory space conditions. In this context, the OS moves (*swaps out*) the least used memory pages to a pre-configured space on the hard disk called *paging file*, *swap file* or *swap partition*, and makes the memory available to those applications that need it in that particular moment. *Swap in* is the opposite operation executed by the OS to restore the memory pages back to the RAM, when they are required by the applications. Even though the swapping technique permits software application to use more memory than the physically available, moving memory pages back and forth from RAM to hard disk generally slows down the system execution.

In the SDN context, the fast/size-limited memory is represented by the B/TCAMs of the network elements and the slow/large memory can be implemented as a database maintained by the SDN controller in the RAM memory of the computer where it is running. Our rationale is to maintain the most matched flow rules in the switches' memory and move the other rules to the database, in a completely transparent way for the network applications running on top of the controller.

We can summarize the contributions of this work as follows:

- **Design** of a platform-independent memory swapping mechanism for SDN controllers, as part of a more powerful Memory Management System, whose aim is to optimize the usage of switches' memory, transparently to SDN network applications.
- **Implementation** of the swapping mechanism for the ONOS platform [5]. ONOS is an advanced SDN controller that provides the required services and APIs for the implementation of the swapping mechanism, as identified in [3]. Unlike other controllers like Ryu [6], where the table full condition must be handled by the SDN applications, ONOS hides these low-level details and holds the flow rules that cannot be installed in a *pending add* state until there is enough memory space in the switches. We compare the ONOS mechanism against our memory swapping.
- **Validation** of the memory swapping mechanism using real traffic traces. We demonstrate that our prototype optimizes the usage of switches' TCAM memory in terms of free space available for new flow wildcard entries and, therefore, it improves the performance of the network in terms of throughput.

The remainder of this paper is structured as follows: Section II introduces the design of the memory swapping mechanism. Section III illustrates the software architecture and shows its feasibility by describing a prototypical implementation based on the ONOS SDN platform, which is later evaluated in Section IV. We discuss the related work in Section V, while Section VI concludes the paper.

II. Memory swapping design

The memory swapping mechanism automatically frees the TCAM memory of the switches from the least used wildcard flow entries by temporarily moving them to a slower memory. The swapping process consists of two different operations. The first, called *swap out*, is performed when the SDN controller detects that the flow tables are full. In this case the mechanism swaps out the least used wildcard rules to free up TCAM memory space for new entries. Vice-versa, the *swap in* operation restores the swapped out rules when they are needed again by the network device to forward the traffic.

A. Swap out

By default, the *swap out* operation is executed when the SDN controller detects that one or more switches are operating in *table full* condition. In case of OpenFlow-enabled switches, that condition is notified to the SDN controller via a `TABLE_FULL` error message. From OpenFlow 1.4.0 or higher, the swapping mechanism can be also configured to react to `TABLE_STATUS` events with reason `VACANCY_DOWN`, meaning that the remaining space in the flow table has decreased to less than a pre-defined threshold.

Unlike a traditional computer OS virtual memory system, where a page is either allowed to be swapped out or not, the removal of a flow rule from the TCAM may change the semantic of the network. This problem may happen because high priority rules may be dependent on low priority ones. As a trivial example, consider a rule set where the default rule drops any packet and any allowed communication is handled with higher priority rules. When the MMS swaps out an entry of this rule set because of resource pressure, any further packet matching this rule will be then caught by the default rule and dropped, which is an undesirable result. So, whenever we consider a rule as a candidate to swap out, we have to build the (transitive) set of (lower priority) rules which are also affected. The analysis of the rule-dependency problem, which is outside the scope of this paper, has already been tackled by other works such as CacheFlow [7].

Authors of CacheFlow propose an algorithm to incrementally analyze and maintaining rule dependencies. In such a work, a dependency between a child rule C and a parent rule R is defined as follows: if C is removed from the flow table, packets that are supposed to hit C will hit rule P.

The current design of the MMS leverages on CacheFlow's

algorithms to correctly compute the dependencies between rules and, consequently, to *swap out/in* the wildcard entries between TCAM and MMS databases. Please note that, the memory swapping mechanism focuses on wildcard entries to free TCAM memory space. Thus, it does not take into consideration the rarely used exact match rules (which are likely stored in other memories such as DRAM or SRAM), to avoid the risk freeing the wrong memory. Since exact match rules cannot be parents of wildcard rules, swapping only wildcard rules does not create any inconsistency in the switch's flow table.

The pseudo-code in Listing 1 illustrates how the *swap out* process is executed. When an SDN application requests for a flow rule installation, the MMS intercepts the rule, it checks whether the rule contains any wildcard and computes the dependencies with the other wildcard rules already installed in the network and saved in the `flow_db` (lines 2-4 in the Listing). Then, the rule is stored in the `flow_db` database (line 5) along with the dependency chain. Before installing the flow rule, we retrieve the dependency chain of the new rule from the `swap_db` (which stores the rules that have been previously swapped out), to avoid network inconsistencies (line 6). The actual installation of the new rule onto the device is performed at line 7. If the operation returns either a `TABLE_FULL` error or a `VACANCY_DOWN` event, the `swapOut` function is called to free up some space in the switch flow table (line 8) and then the rule is finally installed.

```
1  function installFlowRule(flow_rule fr):
2    if check_wildcards(fr) is True:
3      dep_chain=compute_dependencies(fr,flow_db)
4      fr.addDependencies(dep_chain)
5      flow_db.add(fr)
6      swap_chain=get_dependencies(fr,swap_db)
7      while install_rules(fr,swap_chain) is False:
8        swapOut()
9
10 function swapOut():
11   least_used_fr=get_least_used_fr(flow_db,quota)
12   for fr in least_used_fr:
13     dep_chain=get_dependencies(fr,flow_db)
14     for dep_fr in dep_chain:
15       if dep_fr.timeout is 0:
16         swap_db.add(dep_fr)
17       remove dep_fr
```

Listing 1. Flow rule installation process.

The `swapOut` function gets the least used flow entries from the `flow_db` (line 11), based on traffic statistics collected periodically (as described below in this section). `quota` is the percentage of the installed rules to be swapped out. By default, its value is 20% for all the switches, but this threshold can be tuned dynamically based on the performance of the network. Swapped out flow entries are removed from the TCAM of the devices and from the `flow_db`. Entries with infinite timeout are saved in the `swap_db` (line 16) and automatically restored by the MMS when necessary. Entries with finite timeout are just dropped, based on the assumption that network applications can autonomously restore them, exactly how they would do if those flow entries were naturally expired due to the lack of matching traffic.

A network application may need to uninstall a flow rule. In this case, the MMS automatically deletes the corresponding entry from the `flow_db` and from the `swap_db` (if necessary) and updates the dependency chains of the child entries in those databases.

The ranking of the least used rules is determined by periodically collecting traffic statistics, such as packet and byte counters, for each wildcard rule. For each rule, the complete history of the collected samples is kept in the `flow_db` until the rule is deleted. The classification of the rules is computed with the Exponential Weighted Moving Average (EWMA) algorithm [8], which weights the statistics in geometrically decreasing order so that the most recent samples are weighted more than the oldest samples. This approach avoids erroneous classifications where, for instance, a flow entry periodically matched by a micro-flow has more chances to be swapped out than an entry matched by an elephant flow far in the past, i.e. no more active.

B. Swap in

Swapped out flow rules are automatically re-installed by the MMS onto the network when the switches need them again to forward the traffic, transparently to the SDN applications running atop the SDN controller.
The mechanism is fairly simple: when a switch does not find a match for an incoming flow inside its flow tables, it sends a *new flow* message (e.g. an OpenFlow `PACKET_IN`) to the SDN controller. The MMS intercepts the message before it arrives to the applications and checks whether the flow matches any of the swapped out rules in the `swap_db` database (line 2 in Listing 2). If the MMS finds a match, it automatically reinstalls the rule along with its dependency chain into the switch's memory (lines 3-5), otherwise the *new flow* message is released to the other processes of the SDN controller, which eventually relays it to all the listening applications.

```
1 function swapIn(packet pkt):
2   fr=swap_db.get_rule(pkt)
3   if fr is True:
4     swap_chain=get_dependencies(fr,swap_db)
5     install_rules(fr, swap_chain)
```

Listing 2. Swap in process.

III. Software architecture

In [3] we presented the concepts behind the MMS and we listed the requirements for its implementation as a component for a generic SDN controller. Specifically, the MMS requires a number of services and interfaces to interact with the network devices and to accomplish the memory management operations. Since not all the SDN controllers meet the requirements, we started our development work by implementing the *memory deallocation* function for ONOS [4]. In this section, we recall the building blocks of the MMS architecture and we map them into ONOS with specific focus on the requirements for the *memory swapping*.

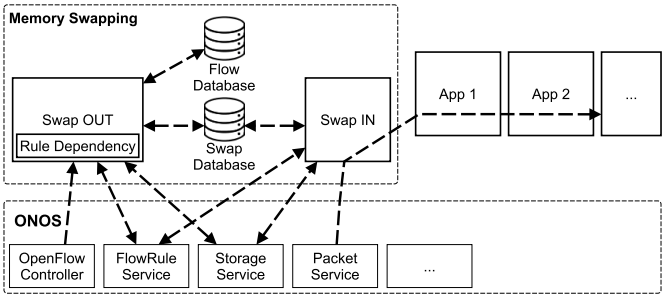


Fig. 1. The memory swapping in the context of the ONOS platform.

A. Interaction with the controller

The *memory swapping* requires read/write access to the flow tables of the switches. It must keep track of all the flow entries installed in the network along with their statistic counters. Moreover, it must be able to intercept some events generated by the network, such as the `TABLE_FULL` error and the `TABLE_STATUS` event with reason `VACANCY_DOWN`, used to trigger the *swap out* process, and the notification of new flows (i.e., `PACKET_IN` in OpenFlow), used by the *swap in* process to re-install any previously swapped out entry matching the new flow with all its dependencies (as explained in Section II-B).

As shown in Figure 1, in ONOS such functions are accomplished by four different interfaces called: *OpenFlowController*, *FlowRuleservice*, *StorageService* and *PacketService*. Follows a description of such interfaces and how they are used by the memory swapping mechanism.

FlowRuleService [9]. Interface for installing/removing flow rules into/from the network and for obtaining updates on those already installed. The MMS is also registered as a listener to this interface to: (i) intercept all the flow rules installed by the SDN applications and (ii) get the statistic counters of the installed flow entries, as soon as such statistics are made available by ONOS which collects them every 5 seconds. This information is used by the memory swapping mechanism to recognize the least matched entries which are swapped out in case of full TCAM.

OpenFlowController [10]. Abstraction of the OpenFlow controller. It is used for obtaining OpenFlow devices, for sending OpenFlow messages to them and to register/unregister listeners on OpenFlow events. Specifically, in the current version the MMS registers as a listener to this interface to get the `TABLE_FULL` error message. We plan to add the support for `TABLE_STATUS` events in the next releases.

StorageService [11]. ONOS is a distributed SDN controller platform. An ONOS cluster comprises one or more ONOS instances, running the same set of modules and sharing network state with each other. In that respect, MMS internal databases are based on the *StorageService* interface which ensures a consistent state of the databases across all the instances of a ONOS cluster.

PacketService [12]. Service for intercepting the control messages generated by the switches in case of table miss events. As part of the registration process to this service, listeners (SDN applications as well as the MMS) specify a priority value which determines the order for processing the event. Lowest is the value, earliest the listener receives the message. The MMS registers with priority value 0 (the lowest), as required for the implementation of the *swap in* function.

B. Building blocks

Flow Database: The MMS implements an internal *Flow Database* to replicate the information contained in the flow tables of network devices. To do so, the MMS is registered as a listener to the *FlowRuleService* to intercept and collect the new rules installed by the network applications. The database is implemented using the *EventuallyConsistentMap* [13] distributed primitive, a data structure provided by ONOS *StorageService* which provides high read/write performance. The structure of the data stored in the database extends the ONOS *FlowRule* to contain: (i) current statistics counters of each flows rule, (ii) the whole history of statistic counters and (iii) the list of parent rules, based on the computation of the CacheFlow algorithm [7].

Swap Database: It is implemented as an *EventuallyConsistentMap* containing all the swapped out flow rules. In the first prototypes, the *Swap Database* was obtained with just a flag in the *Flow Database* indicating whether the flow rule was swapped out from the switches' memory. However, as the number of swapped out rules is low compared to the total amount of entries in the *Flow Database*, we realized that a dedicated database for the swapped out rules was a better idea to minimize the lookup time and to reduce the latency introduced by the *swap in* process when inspecting the *new flow* messages.

Blocks **Swap OUT** and **Swap IN** in Figure 1 represent the two main processes of the memory swapping mechanism.
The *swap out* process is in charge of freeing the switch's TCAM by moving the least matched flow entries to the RAM memory of the machine where the SDN controller is running. This process is configured to react to `TABLE_FULL` errors received via the *OpenFlowController* interface. In our ongoing work, we plan to add support for `VACANCY_DOWN` table status notifications, which will allow the MMS to get an early warning and to execute the swapping process before getting the table full.
The *swap out* process is divided into the following steps: (i) the MMS retrieves from the *Flow Database* all the wildcard entries associated to the switch that generated the `TABLE_FULL` alarm, (ii) the flow entries are sorted based on the average number of matching flows as computed by the EWMA algorithm, (iii) finally the least matched rules are removed from the TCAM with all their dependent rules. The copies of such rules maintained in the *Flow Database* are moved to the *Swap Database*.
Based on our experiments (cf. Section IV), we swap out the 20% of the whole TCAM content. However, this value can be tuned switch by switch at runtime based on the performance of the network, i.e. it should be increased in case of frequent

TABLE_FULL events or decreased in case of too many rules re-installed into the network by the *swap in* process after being swapped out.

IV. Performance Evaluation

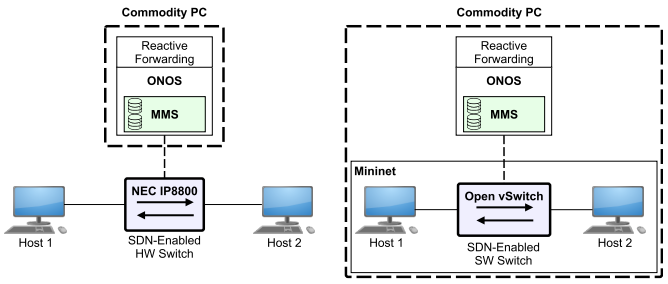
We evaluate the memory swapping mechanism by considering two metrics: (i) average end-to-end throughput and (ii) TCAM space available for new flow entries. The effectiveness of the proposed approach is measured by comparing the results observed with and without the memory swapping using flow table of different sizes.

A. Test methodology

Experimental setup. For the experiments, we use both hardware and software OpenFlow-enabled switches. In the first case we use the NEC IP8800 [14] (Figure 2a), while in the second, Mininet [15] and Open vSwitch (OVS) [16] switches (Figure 2b).

We run ONOS and the MMS on a commodity PC equipped with a Intel i7-5600U quad-core CPU running at 2.60GHz and 16GB of DDR3 memory working at 1600Mhz. This machine is connected to the NEC switch via Gigabit Ethernet for the OpenFlow control channel. Two physical hosts are connected to the switch via Gigabit Ethernet to inject network traffic during the evaluation. For the test with software switches, the commodity PC also hosts Mininet configured with a single OVS-based switch and two virtual hosts attached to it.

We use publicly available SMTP and HTTP traffic traces from [17]. The first one produces 23 new flows per second on average when considering Layer 3 fields, while the second trace produces 65 new flows per second on average.



(a) NEC IP8800 setup. (b) Open vSwitch setup.

Fig. 2. Experiment setups for the evaluation.

Context. The result of the evaluation depends on three main aspects: (i) the flow table size of the switch, (ii) the flow rate of the traffic trace, i.e., number of new flows per second, and (iii) how fast the corresponding flow entries expire. Thus, we expect our memory swapping mechanism to be more effective with small flow tables and high rate of flows controlled using flow entries with high expiry timeouts. To stress the memory swapping, we started with the HTTP trace, but unfortunately we experienced many disconnections of the OpenFlow channel due to CPU overload of our NEC switch. Due to this, we were forced to limit the evaluation with the hardware switch to just the SMTP trace.

The TCAM of our NEC IP8800 can host up to 1500 flow entries, but we are interested to understand the effectiveness of the memory swapping mechanism when varying the size of the flow table. For this reason, we configure the OVS switch at different flow table sizes, from 1500 entries (like our NEC), to 2000 entries like a HP 8200/5400 [18]. In this case, we use the HTTP traffic trace.

Methodology. We compare the memory swapping implementation for ONOS described in Section III with the default ONOS memory management system. By default, ONOS holds in *pending add* state the flow rules that cannot be installed until there is enough memory space in the switches. We demonstrate that our mechanism is better in terms of (i) memory space available in the switches for the new entries, and (ii) performance of the network measured in terms of end-to-end throughput.

The experiments are executed under the following conditions:

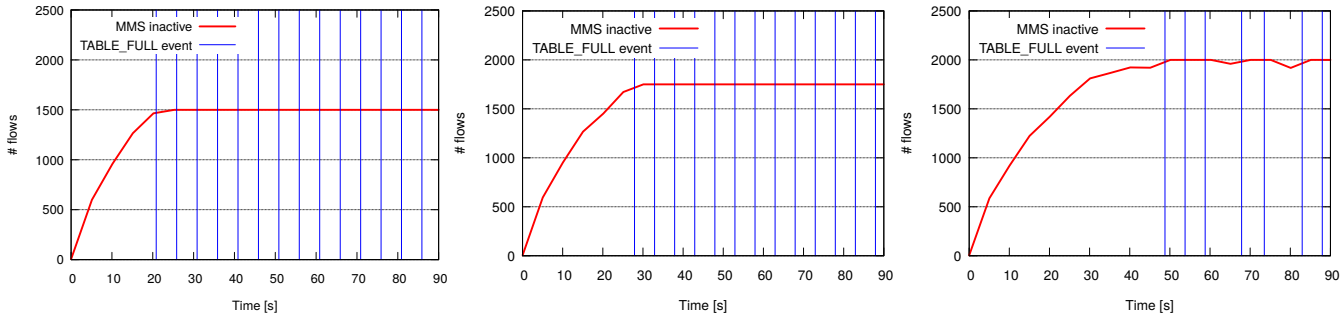
- At time 0, one of the hosts (e.g., `Host1` in Figure 2) starts injecting the traffic trace into the switch (either hardware or software). At this point in time, the flow table of the switch is empty.
- The switch is controlled by ONOS via the *Reactive-Forwarding* application [19]. This application reactively installs a flow entry in the switch for each incoming new flow.
- The application is configured to generate wildcard flow entries with the only IP source and destination addresses specified. The application also randomly assigns either infinite or 10 seconds idle timeouts to the flow entries.
- Dependencies between flow table entries are synthetically generated based on pairs of random IP subnet masks and priorities. Thus, we configured the forwarding application to randomly apply different levels of priorities combined with different submasks by following the longest prefix match principle, where the longer the subnet mask, the higher the priority.

B. Results and discussion

The most tangible benefit of our memory swapping mechanism is the increase in the end-to-end throughput measured at the destination host (e.g., `Host2` in Figure 2). Table I summarizes the results obtained with the NEC hardware switch using the SMTP traffic trace (23 new flows per second), and with the OVS software switch using the HTTP trace (65 new flows per second).

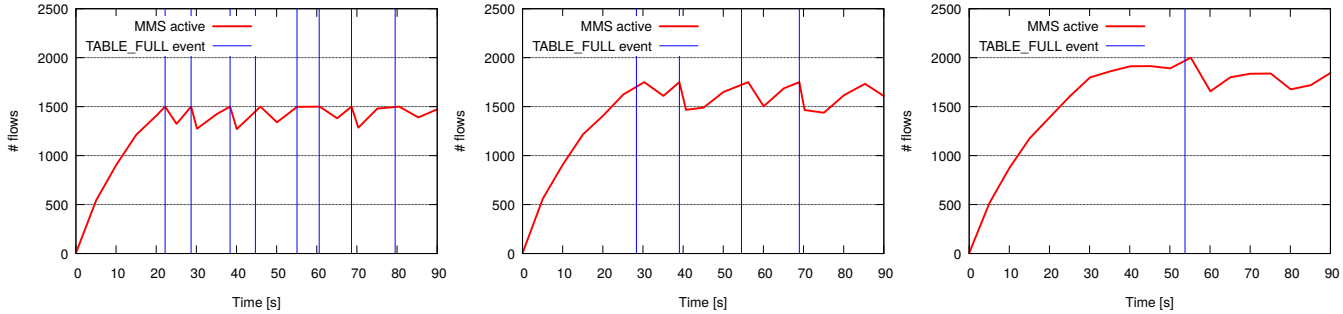
TABLE I
Average throughput.

Flow Table Size	Traffic trace	Throughput MMS Active [Kbps]	Throughput MMS Inactive [Kbps]
1500 (NEC)	SMTP	19.85	15.81
1500 (OVS)	HTTP	80.17	58.97
1750 (OVS)	HTTP	85.63	72.38
2000 (OVS)	HTTP	86.54	81.13



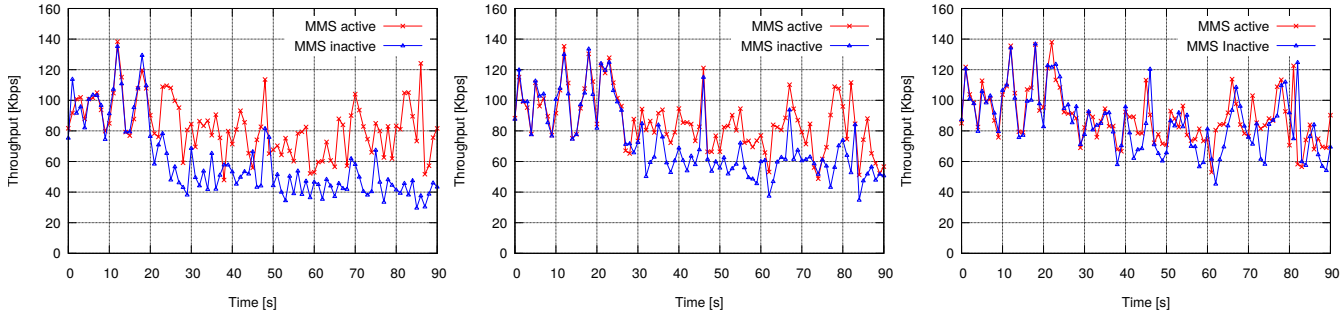
(a) Flow Table size: 1500 rules. (b) Flow Table size: 1750 rules. (c) Flow Table size: 2000 rules.

Fig. 3. Number of installed rules without memory swapping at different flow table sizes.



(a) Flow Table size: 1500 rules. (b) Flow Table size: 1750 rules. (c) Flow Table size: 2000 rules.

Fig. 4. Number of installed rules with memory swapping at different flow table sizes.



(a) Flow Table size: 1500 rules. (b) Flow Table size: 1750 rules. (c) Flow Table size: 2000 rules.

Fig. 5. Throughput comparison with and without memory swapping at different flow table sizes.

Specifically, when using the memory swapping we measure a throughput increase from 15.81 Kbps to 19.85 Kbps (21% on average) with the NEC switch and the SMTP trace. When using the software switch and the HTTP trace, we observe different performance depending on the size of the flow table. We measure throughput increases of 26%, 15% and 6% with flow table sizes of 1500, 1750 and 2000 flow entries respectively.

Results obtained with the OVS-based software switch are also reported in Figures 3, 4 and 5. Vertical blue lines in Figures 3 and 4 represent the `TABLE_FULL` error messages sent by the switch operating in full table condition to ONOS, when ONOS tries to install the pending rules. By default, this installation process is automatically performed every 5 seconds if the *pending add* queue is not empty. Please note that, for the sake of readability, we represent only one error message for each attempt.

Without the memory swapping, the flow table is constantly full and every time ONOS tries to empty the *pending add* queue, it gets a `TABLE_FULL` error (Figures 3a and 3b). The problem is partially mitigated when the software switch is configured with a larger flow table (Figure 3c). In this case, the number of flow rules which are evicted for expiring timeout is often sufficient to make space for the rules waiting in the *pending add* queue. Conversely, the memory swapping process frees the 20% of the TCAM capacity in reaction to `TABLE_FULL` errors (Figure 4). More precisely, the least matched entries are moved to the *Swap Database* and the free space in the TCAM is used by

ONOS to install the rules in *pending add* state and, possibly, the new rules the *ReactiveForwarding* application generates to control new flows. The benefits of the memory swapping are demonstrated by the reduced number of `TABLE_FULL` errors, as shown in Figure 4, and by the increased performance in terms of end-to-end throughput at different flow table sizes, as shown in Figure 5 and also summarized in Table I.

Finally, we observe that the effectiveness of our mechanism increases when the ratio between the flow rate and the flow table size increases. In this respect, recall that SDN platforms like ONOS are designed to scale to large networks, where the flow arrival at the switches can be in the order of thousands flows/sec [20]. Thus, we conclude that a memory management mechanism like the one presented and validated in this paper, can help such SDN environments to operate efficiently without requiring network devices mounting large, expensive and power hungry TCAMs.

V. Related Work

There is a vast array of work related to the TCAM memory utilization and optimization in SDN. We classify the most relevant techniques in five categories:

Flow rule Caching: Like the MMS, CacheFlow [7] is a *Virtual Memory* mechanism that gives SDN applications the illusion of an arbitrarily large switch memory. In CacheFlow, the additional memory is provided by software SDN switches which are attached to the datapath and implemented as software agents in commodity server-class hardware or directly in the hardware switches. However, while the latter approach requires modifications of the switches' firmware, the first imposes strict constraints to the network, as CacheMaster, the component that hosts the software switches, must communicate with the hardware switches via either single-hop Layer-1 connectivity or Layer-2 tunnels. Moreover, CacheFlow does not tackle the *table full* case, i.e. the critical situation when the TCAM of one or more hardware switches is full, which the MMS covers by design.

Exact match rules: Authors of DomainFlow [21] leverage exact match rules to overcome the limited capacities of the TCAM memories, as exact match rules are saved in binary memories (like briefly explained in the Introduction). DomainFlow keys ideas are: (i) use exact matching where possible and (ii) split the network into sections to allow exact matches to be used more often. DevoFlow [20] propose a modification of the OpenFlow model by introducing a new action type in form of the *clone* flag. If the flag is set, the switch clones wildcard rules with exact match rules that are saved in the exact match flow table, i.e. saving TCAM memory space. However, DevoFlow imposes modifications of both OpenFlow protocol and switches' firmware and, like DomainFlow, it does not address the limitations imposed by the size of the TCAM.

Idle timeouts: SmartTime [22] uses adaptive heuristic to compute idle timeouts for the flow rules which results in optimal utilization of the TCAM memory. SmartTime pro-actively evicts flow rules with finite timeout in a random manner when the TCAM utilization crosses a pre-defined threshold. However, SmartTime does not cover the common case in which pro-active rules with infinite timeouts are used to control the traffic (e.g. cloud orchestrators like OpenStack/Neutron pro-actively install all the rules to create the virtual network topologies).

Flow table compression: Tag-in-Tag [23] proposes a technique to replace the OpenFlow entries stored in the TCAM memories with two layers of simpler and shorter tags. However, Tag-in-Tag requires changes in the packet header and in the switches' firmware to correctly handle the tags. Authors of [24] leverage OpenFlow's wildcards to reduce the memory utilization in the specific scenario of Border Gateway Protocol (BGP) routing tables.

Flow rule placement optimization: Several recent works propose algorithms or mechanisms for an optimal rule placement across the network, with the aim of saving TCAM memory space ([25], [26], [27], [28]). However, none of them tackles the *table full* case.

VI. Conclusions

In this paper we presented a novel *memory swapping* mechanism for SDN controllers that aims at improving the performance and the reliability of the network. The mechanism, which is part of a sophisticated SDN Memory Management System, ensures that the flow entries needed to smoothly manage the network, i.e., the most frequently matched and the most recent ones, are stored in the fast TCAM memory of the switches even when the space on such memory reach its limit. Conversely, the least matched and relevant entries are not evicted, as suggested by other approaches, but moved to larger (but slower) memories, such as the RAM memory of the machine where the controller is running, and automatically restored in the TCAMs once they are again required by the network.

The memory management functions are implemented based on the following design principles: (i) transparency to SDN applications and switches (statistic counters of flow entries are also preserved) (ii) applicability to complex deployment scenarios, including carrier and Internet Service Provider (ISP) networks, (iii) compliance with standard control protocols, without imposing any modification of the switches' firmware.

We validated the swapping mechanism with real traffic traces, by measuring the memory space available on the TCAM for new entries and the end-to-end average throughput. As expected, the benefits of our mechanism increase when the ratio between the flow rate and the flow table size increases. Such a result may raise concerns about the possible obsolescence of the proposed approach thanks to technological advance in the electronics industry which will likely result in larger and, at the same time, less expensive TCAMs. On the other hand, the continuous increase of the network traffic rate, especially due to the significant growth of mobile traffic, makes us believe that our work will be relevant over a long term.

ACKNOWLEDGMENTS

The work presented in this paper has been partially sponsored by the European Union through the FP7 project NetIDE, grant agreement 619543.

REFERENCES

[1] 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.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] "OpenFlow Switch Specification 1.5.1." [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf

[3] R. Doriguzzi-Corin, D. Siracusa, E. Salvadori, and A. Schwabe, "Empowering Network Operating Systems with Memory Management Techniques," in *IEEE/IFIP Network Operations and Management Symposium*, Istanbul, Turkey, 25-29 Apr. 2016.

[4] A. Marsico, R. Doriguzzi-Corin, M. Gerola, D. Siracusa, and A. Schwabe, "A Non-disruptive Automated Approach to Update SDN Applications at Runtime," in *IEEE/IFIP Network Operations and Management Symposium*, Istanbul, Turkey, 25-29 Apr. 2016.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, Chicago, Illinois, USA, 22 Aug. 2014.

[6] Ryu SDN Framework. [Online]. Available: http://osrg.github.io/ryu/

[7] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks," in *Symposium on SDN Research*, Santa Clara, CA, USA, 14-15 Mar. 2016.

[8] J. M. Lucas, M. S. Saccucci, R. V. Baxley, Jr., W. H. Woodall, H. D. Maragh, F. W. Faltin, G. J. Hahn, W. T. Tucker, J. S. Hunter, J. F. MacGregor, and T. J. Harris, "Exponentially weighted moving average control schemes: Properties and enhancements," *Technometrics*, vol. 32, no. 1, pp. 1–29, Jan. 1990.

[9] "ONOS Interface FlowRuleService." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/net/flow/FlowRuleService.html

[10] "ONOS Interface OpenFlowController." [Online]. Available: https://github.com/opennetworkinglab/onos/blob/onos-1.6/protocols/openflow/api/src/main/java/org/onosproject/openflow/controller/OpenFlowController.java

[11] "ONOS Interface StorageService." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/store/service/StorageService.html

[12] "ONOS Interface PacketService." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/net/packet/PacketService.html

[13] "ONOS Interface EventuallyConsistentMap." [Online]. Available: http://api.onosproject.org/1.6.0/org/onosproject/store/service/EventuallyConsistentMap.html

[14] "NEC IP8800 OpenFlow Networking." [Online]. Available: https://support.necam.com/SDN/ip8800/

[15] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-based Emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, Nice, France, 10-13 Dec. 2012.

[16] The Open vSwitch Project. [Online]. Available: http://openvswitch.org/

[17] A. Dainotti, A. Pescapé, P. S. Rossi, F. Palmieri, and G. Ventre, "Internet traffic modeling by means of Hidden Markov Models," *Computer Networks*, vol. 52, no. 14, pp. 2645 – 2662, 2008.

[18] "HP OpenFlow 1.3 Administrator Guide," Hewlett-Packard, p. 15, Jun. 2015, rev. 2.

[19] "ONOS ReactiveForwarding." [Online]. Available: https://github.com/opennetworkinglab/onos/blob/onos-1.6/apps/fwd/src/main/java/org/onosproject/fwd/ReactiveForwarding.java

[20] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, Toronto, Ontario, Canada, 15-19 Aug. 2011.

[21] Y. Nakagawa, K. Hyoudou, C. Lee, S. Kobayashi, O. Shiraki, and T. Shimizu, "DomainFlow: Practical Flow Management Method Using Multiple Flow Tables in Commodity Switches," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, Santa Barbara, California, USA, 9-12 Dec. 2013.

[22] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective Switch Memory Management in OpenFlow Networks," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, Mumbai, India, 26-29 May 2014.

[23] S. Banerjee and K. Kannan, "Tag-In-Tag: Efficient Flow Table Management in SDN Switches," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, Rio de Janeiro, BR, 17-21 Nov. 2014.

[24] W. Braun and M. Menth, "Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-Based Software-Defined Networking," in *2014 Third European Workshop on Software Defined Networks*, 2014.

[25] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, p. 351, 2010.

[26] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too Many SDN Rules? Compress Them with MINNIE," in *2015 IEEE Global Communications Conference (GLOBECOM)*, San Diego, CA, USA, 6-10 Dec. 2015.

[27] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Optimizing rules placement in openflow networks: Trading routing for better efficiency," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, Chicago, Illinois, USA, Aug. 2014.

[28] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM 2013 - IEEE Conference on Computer Communications*, Torino, Italy, 14-19 Apr. 2013.