A FRAMEWORK TO RAPIDLY TEST SDN USE-CASES AND ACCELERATE MIDDLEBOX APPLICATIONS

Rajesh Narayanan, Geng Lin
Dell Inc.
{N_Rajesh, Geng_Lin}@dell.com

Affan A. Syed
NUCES, Islamabad Campus
affan.syed@nu.edu.pk

Saad Shafiq, Fahd Gilani
xFlow Research
first.last@xflowresearch.com

Abstract*—Software-defined networking (SDN) is envisioned to provide a centralized interface to programmatically manage networking elements. However, despite its conceptual simplicity, current switch and SDN architectures have poor performance with little support to innovate and test novel SDN applications. We propose an application extensibility framework that allows researchers to build new SDN applications without requiring modification to the OpenFlow-based plumbing available today. We also employ both hardware and software packet processing capabilities of switching elements by offloading intensive per-packet processing onto the switch processing pipeline using dynamically loadable packet processing modules (PPMs). Our architecture thus allows flexibility in the type of applications alongside high switching performance. We believe that our architecture will unleash the potential of SDN by inspiring the SDN "killer app". We evaluate our framework using an encryption middlebox application and show a two orders-of-magnitude improvement over an implementation using the current SDN architecture when using hardware offload blocks.

***Index Terms*— Middlebox, Software Defined Networking**

I. INTRODUCTION

The recent interest in software-defined networking (SDN) has origins in the desire to allow experimental protocols tested at the scale of production networks [1]. This desire led to the formation of the Open Networking Foundation (ONF) and a groundswell of support from a diverse set of players ranging from switch manufacturers to data centers and enterprise users.

The current SDN architecture builds upon the OpenFlow protocol that decouples the control and data forwarding functions of a traditional switch into two separate logical entities [1]. Thus it proposes using logic at a centralized controller that then employs OpenFlow to populate forwarding tables on simple, but fast, forwarding/switching elements throughout the network.

However, despite its disruptive potential, the industry remains skeptical of SDN's ability to provide a truly novel and practical solution that existing networking framework cannot provide. We believe that there are **three** reasons for this skepticism.

First, currently the bar to writing robust SDN applications remains quite high as there is no ability for developers to reuse code and build upon existing work. The resulting small nucleus of SDN application developers reduces the probability of building truly innovative and potentially game-changing applications.

Second, several interesting applications requiring deep or per-packet processing cannot work at line-rates due to switch-to-controller latency being reported on the order of ten-to-hundreds of milliseconds [2].

Lastly, we observe that there is currently no way for SDN applications to use packet processing hardware acceleration blocks present in several network processing units (NPUs) [3] [4] [5]. We believe that these hardware blocks provide much needed additional muscle to the switching element that will be essential for many interesting SDN applications.

In this paper we present an **SDN application extensibility framework (AEF)** to alleviate these deficiencies. Our framework provides SDN application developers the ability to reuse *modules* of written code. This concept is similar to the usage of libraries and SDK's provided by third parties that accelerates the standard software development cycle. This framework also allows us to instantiate code at the switching element for fast processing of flows. The framework is modular and extendable, inasmuch that code output can be easily chained together for greater functionality. Furthermore, we extend the framework to provide a transparent mechanism for using hardware acceleration present on commodity switches and NPUs.

Most importantly we design our framework to provide all these capabilities without requiring *any changes to the OpenFlow protocol*. This design allows us to leverage existing and future development and expertise in OpenFlow while accelerating the control plane in an orthogonal fashion. To recap, our AEF removes the controller-switch latency issues, allows utilizing hardware offload blocks, and has the ability to reuse robust SDN code.

In the rest of this paper we begin by first motivating the need for a new SDN framework (Section 2). We next provide an overview of our framework and give details regarding its

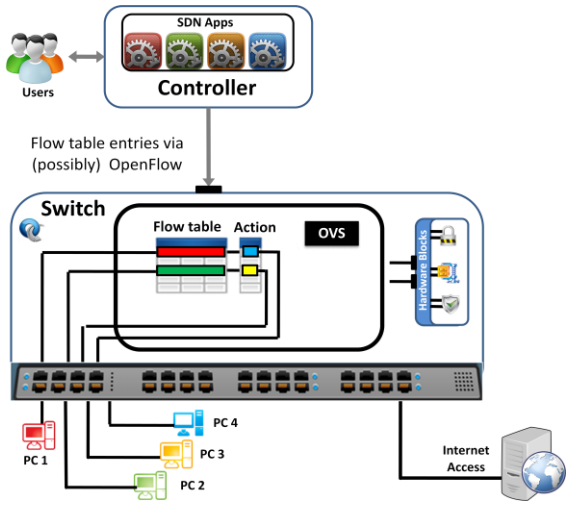

Figure 1: Current SDN Architecture. OVS represents a generic OpenFlow compliant component on a switch.

implementation (Section 3). We then provide evaluation results detailing the benefits of using our framework (Section 4). We end by discussing the architecture and its extensions that are part of our future work.

II. Motivating the SDN Application Extensibility Framework

The core idea of our extensibility framework is to *accelerate the control plane for middlebox applications* in an OpenFlow-enabled network by adding arbitrary packet manipulation logic *at the* switch itself. The need arises with current SDN architecture limiting the amount of in-network packet processing possible due to the inherent switch-to-controller bottleneck.

In this section we first explain the current architecture and the motivating drawbacks therein. We end by specifying the design goals of our extensibility framework.

A. Current Architecture and its Drawbacks

The current *atomic* unit of OpenFlow-based SDN architecture comprises a hardware switching fabric whose flow table entries are remotely manipulated using a controller (Figure 1). The implementation of the OpenFlow protocol, for example Open vSwitch (OVS) [6], allows a controller (possibly remote) to set flow matching rules based on the SDN application[1] logic implemented at the controller. When flows match, a set of simple packet header manipulation and forwarding decisions can be executed by the OpenFlow implementation.

We observe that the current architecture has three drawbacks that will motivate the design goals of our application extensibility framework.

1. Any in-network, packet processing capability (for example, IDS/IPS, encryption, TCP de-dup [7], [8]) requires sending all packets of the flow to the controller. The inherent switch-to-controller latency, reported to be in the order of ten-to-hundreds of milliseconds [2], however renders such an implementation untenable at line-rates. While current deployments route flows to middlebox appliances on the same rack, it is not feasible to always have such a short latency middlebox present. Furthermore, our results show significant throughput improvement over even such an implementation using a middlebox on the same rack. The core reason flows are routed outside of the switch stems from a limited action set provided by OpenFlow. This action set is applicable to do basic packet header manipulation and filtering, but unable for more complicated processing of the packet itself.
2. Currently the bar to writing any SDN application is quite high for people with novice to moderate networking skills. One reason is the need to understand all capabilities of the OpenFlow protocol and the need to code *all* networking logic. Another is the inability to reuse mature and stable code, like libraries used in software development. We believe that just as libraries accelerate software development, having a similar capability will accelerate SDN application development and thus motivate early adoption of the SDN concept.
3. There is no support in the current architecture to utilize hardware accelerators on commodity switching hardware. Several switch and NPU manufacturers support in-network traffic engineering in the form of hardware offload blocks [3] [4] [5]. These are traditionally used to provide line-rate traffic engineering applications, but the current SDN architecture renders them unusable.

We note that several complementary attempts have been made to provide similar capability to program the forwarding plane (e.g. Switchblade [9], and Cisco OnePK [10]). These efforts provide extensibility but are generally not coherent with the OpenFlow standardization process.

B. Design Goals of our Framework

We now define the design goals of our SDN application extensibility framework (AEF), which essentially aims to remove the shortcomings in the current SDN architecture we describe in the previous section.

1. We need to provide a way to accelerate the control-plane for line-rate packet processing using a *semantically transparent* mechanism to enrich the OpenFlow action set. The semantic transparency allows applications to be built using existing controllers [11] [12] [13], leveraging the tremendous

[1] The common definition of an SDN application as used in this paper, limits to code written at *only* the controller.

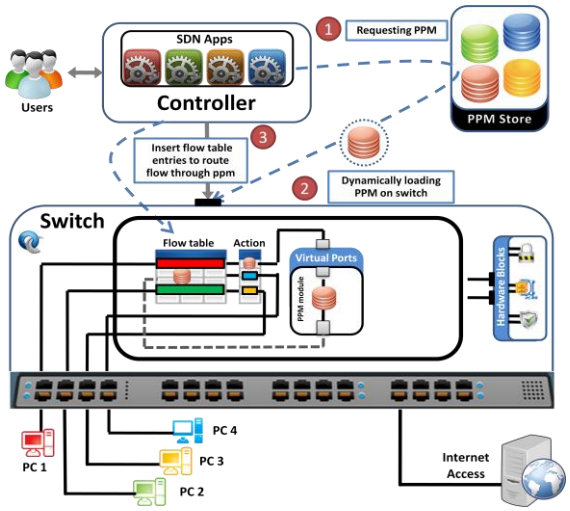

Figure 2: The SDN application extensibility framework accelerates the control plane for middlebox applications by adding arbitrary packet manipulation logic at the switch.

amount of development effort in SDN orthogonal to our framework.

2. Our framework needs to support the ability to reuse mature SDN code modules, which we will call Packet Processing Modules (PPM), by providing an ability to dynamically load and use them in any SDN application.
3. Our framework should also provide a mechanism for SDN applications to use hardware offload blocks that can be provided by a switch or NPU to better leverage the hardware acceleration potential.

We realize that what we are advocating is blurring the clean split between the control and data plane of network switching elements. However, we observe that the nature of packet processing and middlebox applications cannot be captured by the *logically* clean separation of the control and data plane. Networking applications that currently require deep and per-packet manipulation (e.g. firewalls, encryption, and compression) are intrinsically in a *gray area* between these two planes. The current SDN architecture forces the realization of these applications only via per packet inspection at the controller or, if available, routing through networking appliances sharing the same ToR switch. This switch-to-controller or even switch-to-appliance latency is, however, a practical impediment to the sucessful realization of these applications which are an industry need.

III. Implementing the Application Extensibility Framework

We now present our application extensibility framework that is guided by the design goals described in Section II.B.

We start by describing an overview of our framework, and then describe in detail the different modules that constitute this framework.

A. Overview of the Extensibility framework

We propose an extensibility framework which has three key features.

1. Our framework allows a mechanism for packet processing code to be loaded dynamically onto a switch in the form of modules (packet processing modules or PPM).
2. Our framework abstracts the installed PPM as another port on the switch. Since applications on the SDN controller view the data plane just as a switching fabric, they can then use the packet processing code by simply adding a flow-table entry for the designated flow.
3. The framework embraces the concept of a separate library or store containing robustly developed and mature SDN logic in the form of PPM. These PPMs can be used by average SDN developers to rapidly build innovative SDN applications.

Figure 2 shows an overview of our proposed framework. A user developing SDN applications using our framework can offload per packet inspection to the switch downloading the packet processing module (PPM). Our framework makes this module, once at the switch, accessible as a virtual port on the switch. These virtual ports will now be available on the SDN controller. When the SDN application logic determines the need to use the PPM (by inspecting the first packet of the flow at the controller), it needs to add two entries to the flow-table.

The first entry forwards all packets to the newly created *virtual* port on the switch. All subsequent packets of that flow will then be passed through the PPM and manipulated at line rate. The output of the PPM will then pass through the flow-table again, where the *second entry* written by the controller app will apply the actual control plane forwarding decision.

Our framework, therefore, provides an ability to arbitrarily enrich the action set of an OpenFlow compliant switch, *without* altering the protocol specification itself. We believe that this capacity of our framework makes the AEF an immediately realizable solution leveraging existing SDN experience and deployments.

We observe that the PPM can be either a user written code or fetched from a repository of robust and mature code developed externally. This choice represents a tradeoff between *flexible functionality* and *robustness* of the modules. Thus, allowing user to write their module provides a way to export limitless functionality onto the switch forwarding path. On the other hand, the ability to reuse pre-existing modules promotes both robustness and the rapid deployment of SDN applications.

B. Implementation of the SDN Application Extensibility Framework

We now present the implementation of the Application Extensibility Framework (AEF) over the same platform used to develop the Split-Data Plane (SDP) architecture [14].

We therefore used a Dell PowerConnect 7024 platform with a 24x1Gbps switch chip as well as a XAUI interface (Figure 3). The XAUI interface is used to interface a daughter-NPU card, which is a 4-core MIPS and has both encryption and TCP acceleration blocks. We implemented the Application Extensibility Framework (AEF) as software modules over this NPU using the Linux Kernel version *2.6.28*.

In order to demonstrate the control plane acceleration, no rule populated the switch TCAM, and the OVS agent on the switch forwarded all packets to the daughter card over the XAUI interface, bypassing the switch CPU. Destination port information is currently hard-coded; however it can be conveyed over the XAUI interface using the Broadcom HiGig headers supported by both the main switch and the daughter-card [15].

We believe that while the SDP architecture is essential for real-world deployment, the application extensibility framework we describe in this paper is based on an orthogonal argument. We believe that the split-data plane complements the accelerated control plane provided by our architecture; it is however not necessary and was thus removed in our description of the framework in Figure 1-3.

We next describe each component of the implementation in detail.

1) Dynamic loading of Modules

A first goal of our framework is to provide an ability to load modules onto the switch. A key requirement is that this process should not involve a switch reboot as that would make the process untenable for deployment in production traffic.

We use a fairly simple approach where we provide users with SSH-based access to the daughter-card NPU using scripts on the controller machine and the NPU. Our client scripts (*ppm_load)*, interacting with switch-side scripts, provide users with an ability to dynamically select and load PPMs from the client machine. Once uploaded, we first create virtual port(s) (the implementation details of these virtual ports are covered in the next section) that are reported to the user. Then the PPM is executed so that it starts running as a user-space application. In cases where the application is ``*mirrored''* (e.g. encryption or compression), the user can set a flag to simultaneously create two ports.

Since the switch has an OpenFlow implementation (using OVS), the newly created virtual port(s) automatically show up on the controller interface for use by any SDN application code. Our switch-side scripts provide a lightweight load

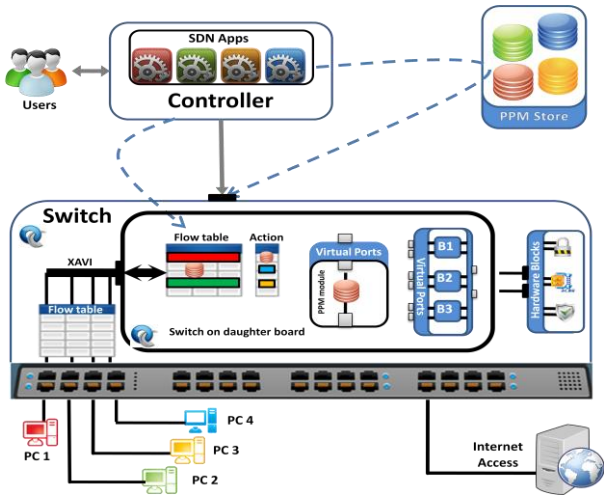

Figure 3: Implementation of AEF over the SDP architecture using a Dell PowerConnect Server with an NPU connected over the XAUI interface

balancing mechanism, fairly loading the PPM on one of the four cores present on our NPU board by setting the corresponding coremask.

The application developer will then write the two new flow-table entries; one to route packets through the newly installed PPM, and the second to forward the modified packets out of the switch port based on the control plane decisions (Figure 2). In our implementation that switch port can be specified using the HiGig headers.

Our framework also provides a mechanism to unload previously loaded PPMs. Thus a user can connect to the switch, view existing PPMs instantiated by that user, and select to unload an individual module by mentioning its name. As a result, any virtual ports previously created are removed. However, care must be taken to remove flow-table entries, using the OpenFlow protocol, prior to such an action.

We note that our solution assumes a level of trust with respect to the execution of externally downloaded code on the switch. We believe that this framework, therefore, shows a natural preference for using robust and verified code from third parties (the PPM store in our figures), as opposed to user-defined code. While we are evaluating mechanisms to improve the security model, the current model is similar to the "walled garden" concept of software security, popularized by Apple iOS ecosystem [16].

2) Implementing the Virtual Port abstraction

A second implementation challenge for our framework was to provide a mechanism to create an arbitrary number of ports at the switch. Furthermore, these *virtual* ports should appear as simply another port on the switch at the controller, thus providing a semantically consistent interface for SDN applications to utilize them in accelerating the control path for packet processing applications.

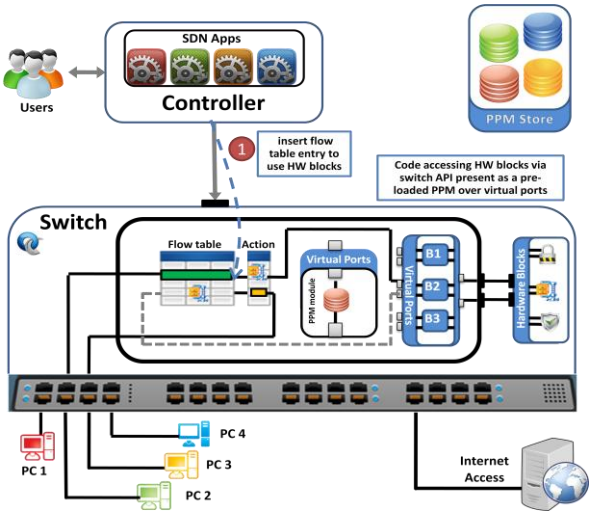

Figure 4: AEF provides a straightforward method to support accessing on-switch hardware offload blocks

We decided to use the TUN/TAP framework to provide this virtual port abstraction due to two reasons. First, the TUN/TAP framework is widely used and supported in the community; thus, using this framework to implement a packet processing module (PPM) is an easy task. The TUN/TAP interfaces appear as regular character devices on the Linux kernel. Developers of PPM, running in user space, can simply open and interact with the incoming packet flows using traditional system calls for *open(), read(),* and *write()*. OpenVPN, a commonly used open source VPN client, uses the TUN/TAP interfaces in a similar fashion to encrypt and tunnel traffic between the end points of a VPN [17].

A second reason for choosing the TUN/TAP framework is its ability to scale to large number of PPMs on a switch. We evaluated the use of internal hardware loopback interfaces; however, these are finite and their number varies for different NPUs. Thus, for example, Cavium Octeon Series processors have between 2-16 internal loopback interfaces [4]. Therefore, even though TUN/TAP devices incur greater latency than loopback interfaces due to additional translations through the protocol stack, their ability to scale tilts the balance in their favor for our implementation. In effect, our current implementation exchanges flexibility for efficiency. We are, however, also evaluating combining the efficiency of hardware interfaces with the flexibility of TUN/TAP interfaces for our purposes as part of future work.

In our implementation we use the L2 TAP interface to create virtual ports. This L2 port is then added to the OVS bridge and appears on the OpenFlow controller. Any new rule that forwards packets to this port has the following short packet-walk: OVS bridge routing rules forward the packet (L2 frame) to the TUN device. Users can manipulate these packets appearing at the character device, and write back to the same character device to make the packet exit from the same port. Now, the second flow table lookup is performed to route the packet based on the control plane decision. While not currently implemented, we plan to use HiGig headers to allow choosing the destination switch port from which the packets will egress the switch.

3) Hardware offload blocks as pre-built PPMs

We discussed earlier that several network processor ASICs provide specialized hardware blocks to help the core offload some common in-network traffic engineering functions. Again, OpenFlow-based SDN, with a complete separation of control and data-plane, currently provides no way to use these capabilities present in commodity cards.

We believe that our Application Extensibility Framework (AEF) provides a natural solution to this problem. A first-order approach would be for users to write the code to access these hardware blocks in a PPM and load it into the switch using our framework. However, directly using these device-specific blocks requires knowledge of hardware architecture as well as the API suite to access it. Both these requirements significantly raise the bar to directly program PPMs that are able to use these hardware accelerators.

We lower the barrier to using these hardware blocks by including *pre-loaded* PPMs onto an AEF-compatible switch (Figure 4). These PPMs will be provided by the NPU (and by corollary the offload block) manufacturers. Thus we envision that for every hardware offload block, not only are code samples provided to use these blocks in simple code, but complete PPM modules with the correct interfaces are developed and deployed on an AEF compatible switch.

We propose two ways in which these offload blocks may be used. First, users can simply write flow-table entries to pass traffic through the pre-loaded virtual ports corresponding to each hardware block. This is the simplest mechanism to access just the basic functionality of these resources.

A second approach allows users to open the corresponding TUN interface in their PPM code and use the offload blocks within their application logic. This functionality is for more advanced applications built on top of these hardware accelerators, while still not requiring architecture specific knowledge. Thus an SDN application developer can maintain focus on the networking, and not the device, aspect of an SDN application.

C. Discussing the Extensibility Framework

Our application extensibility framework allows for rapidly developing SDN applications. We see this as a novel way in which the strength of commodity switches and NPUs can be harnessed to seamlessly deliver powerful SDN applications which cannot be deployed in the current OpenFlow-based SDN realizations.

Our framework leads to two natural extensions. First we see that the PPM can naturally be *chained* to allow for a modular approach to developing SDN applications. Thus while an encryption and compression PPM can be built and tested separately, once present these modules can be chained to provide a more powerful realization.

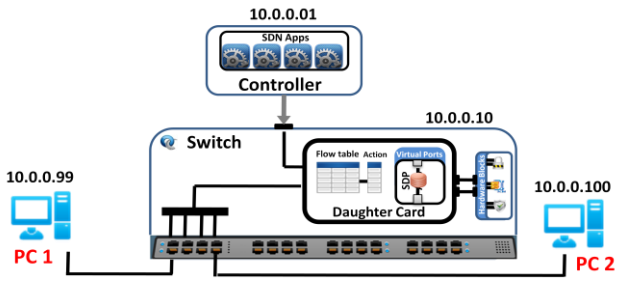

Figure 6: Evaluation Setup using a Dell PowerConnect switch and an NPU-based daughter card

Secondly, we believe that the concept of a PPM can be more powerful if certain parameters can be kept as state. Thus an encryption PPM at the same port can be used by different applications by keeping state in the form of a flow specific key. Such an extension would therefore allow abstract PPM definition that can be generically deployed and used for different instantiations provided by the user.

Beyond these extensions, the concept of a PPM Store leads to a unique capability: experimental evaluation at large scale over heterogeneous networks and deployments. We can therefore envision researchers devising new SDN applications and distributing them through a PPM store. Volunteers and collaborators from across the globe can install these experimental PPMs and perform experiments at a much larger scale than possible at a single facility. Researchers have previously pointed out a similar positive effect occurring in the domain of mobile phone sensing with the introduction of application stores [18].

IV. Experimental Evaluation

We now evaluate the benefits of our application extensibility framework (AEF) in accelerating the control plane and providing greater flexibility in developing SDN applications that require deep packet processing.

We first explain the evaluation setup that we built for performing our experiments. Then we explain our experimental results that show improvement in throughput when using the AEF.

Evaluation Setup

Our AEF has two components; a software PPM implementation of packet processing logic and a similar functional implementation using hardware offload blocks on current network processors and switches.

We therefore evaluate both these capabilities versus the solution possible via the current SDN architecture (Figure 1) where packet processing capabilities are implemented at the controller.

For this purpose, we were restricted to using an application that was supported by the Cavium NPU card that we use in our implementation. Since our daughter card supports the encryption service, we use per packet encryption as an example application to evaluate our framework.

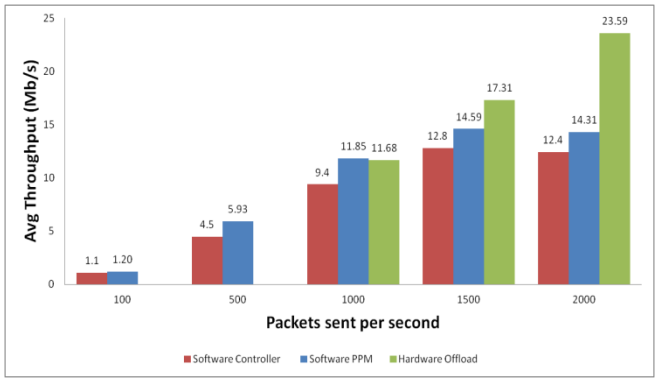

Figure 5: Affect of transmission rate on throughput

The evaluation setup is shown in Figure 6 where we attach two computers, PC1 and PC2, to our implementation of AES (as described in Section III.B). We generate TCP packets using the Ostinato [19] tool at PC1, and use Wireshark [20] to record the reception of packets at PC2. All metrics were evaluated as an average of 100K packets that were generated Ostinato.

In order to evaluate the benefit of using our framework, we construct three different experimental scenarios that correspond to three different packet walks.

1. **Software controller-based Encryption**: This scenario corresponds to what can be implemented using the current SDN architecture. Notice that since our controller is a single hop away, this scenario also captures the currently possible option for packet processing through a network appliance/middlebox that is on the same ToR switch. Here the packets generated follow the PC1→ Dell→ NPU → controller (software encryption) → NPU → Dell→ PC2 path.
2. **Software PPM-based Encryption:** This scenario corresponds to doing encryption using *gcrypt []* inside a PPM instantiated on the daughter card. Here the packet walk is: PC1→ Dell→ NPU (encryption PPM) → Dell→ PC2
3. **Hardware offload Encryption:** This final scenario employs the encryption offload hardware block on the daughter card. Here the packet walk is: PC1→ Dell→ NPU (*hardware* encryption PPM) → Dell→ PC2

A. Evaluation Results

1) Maximum switch throughput

We first evaluate the maximum throughput achievable for our evaluation setup. For this purpose we run a simple application where packets are sent through the SDP switch architecture *without* any packet processing done on them at all. We have a pre-defined rule in the flow-table that precludes going to the controller, so that switching latency is the only bottleneck.

Since both packet size and transmission rate (packets sent per second) affect the throughput, we vary both of these to

Table 1: Application independent maximum throughput for our evaluation setup

Packet Rate (PPS)	Baseline (Mb/s)	Packet Size (Bytes)	Baseline (Mb/s)
3000	34.541	**100**	6.203
4000	45.545	**300**	19.12
5000	54.461	**500**	32.144
6000	65.967	**900**	51.87
7000	75.386	**1100**	70.74
8000	*83.118*	**1300**	83.683
8500	*96.575*	**1500**	96.207

ascertain the maximum throughput achievable in our setup between PC1 and PC2.

Table 1 shows the result of our evaluation. We first observe that the maximum throughput is being achieved at packet size of 1500 bytes. Then using this packet size we vary the packet transmission rate to observe that the maximum throughput of 96.57 Mb/s is achieved at a transmission rate of 8500 packets per second (PPS). Transmission at rates beyond this leads to lowering of throughput, with switch buffers the likely cause of this bottleneck.

2) Comparison of throughput with varying transmission rate

We next compare the throughput of all three implementation scenarios of the encryption application with the purpose of identifying what rates are most suitable for each scenario. Notice that while 8500 PPS is most suitable for the applications where no packet processing is happening, this optimal point should change as packet processing takes different amount of time in our three implementation scenarios.

Figure 5 shows the average throughput result of an experimental setup where we varied the number of packets sent per second being sent by PC1.

We observe two things from this graph. First, the throughput of hardware offload scenario continues to increase with higher transmission rate. In fact, we observe (not shown in the figure) that maximum throughput for hardware accelerated encryption also occurs around the 8500 PPS benchmark set for the plain scenario we explored in the previous section.

The second thing we observe is that the maximum throughput for both the *software PPM* based encryption as well the *software controller* based encryption goes down beyond the 1500 packets/sec mark.

We thus select a transmission rate of 8500 packets/sec for the hardware offload encryption, and a rate of 1500 packets/sec for the other two scenarios. This choice was made, in consideration of the above results, to ensure a fair comparison of the maximum throughput in each scenario.

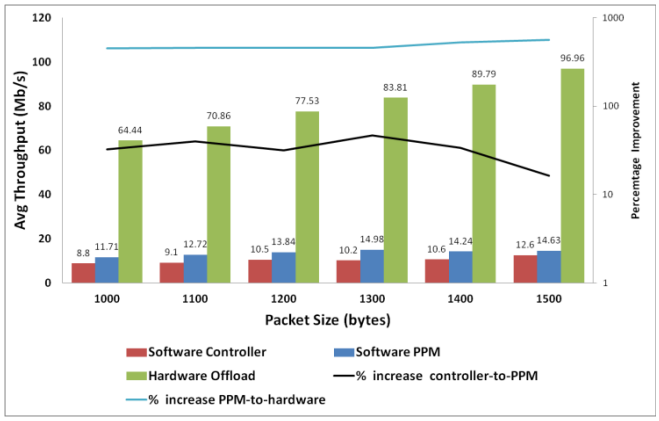

Figure 7: Throughput for encryption application as packet size is changed. Note that the right Y-axis is in log scale

3) Throughput improvement using AEF

We now fairly evaluate, having identified the most suitable transmission rate for each application, the throughput of our three scenarios implementing encryption.

For this purpose we compare both the throughput achieved by our three implementations of per packet encryption as well as the percentage increase in throughput between the different scenarios.

Figure 7 shows the resulting throughput, as packet size varies, and the increase in throughput for two important cases; first, it shows the increase between software encryption when done at the controller with that done at switch using a PPM. Second, it shows the increase between encryption using a hardware offload PPM and a software PPM.

We observe three things from the graph. First, the scenario with software encryption at the controller has, as expected, consistently the worst throughput of all three realizations.

Second, we observe that the increase in throughput while using a software PPM on the switch results in as high as 46.79% increase (1300 bytes). This increase is solely due to the removal of switch-to-controller latency as well as the additional bi-directional stack traversal in the case of encryption done on the controller machine. Furthermore, this scenario also represents the throughput increase when comparing an implementation of packet processing functionality through a switch sharing middlebox appliance as our controller is a single hop away from the switch.

Last, we observe that the improvement when using hardware offload blocks for encryption is 560% when compared with the software PPM implementation. When comparing with encryption at the controller (not shown) the improvement is nearly two orders of magnitude. This last result points to the importance of not only utilizing hardware offload blocks when present, but more importantly it emphasizes the need to have such capability in implementing any network middlebox capability in a SDN environment.

V. Conclusion

In this paper, we present a new SDN application development framework that we believe is essential in enabling network processing and middlebox functionality to be realized in the Openflow-based SDN world. We propose extending functionality on the side of a programmable switch (software, CPU, GPU, or NPU) to load packet processing code in the form of packet processing modules (PPM). This functionality becomes acessible as virtual ports which can be accessed by an SDN application without requiring modification to the OpenFlow protocol. We further provide a mechanism to easily access the hardware offload blocks on the switch. We evaluate our framework to show two orders of magnitude improvement in throughput when using hardware acceleration blocks for an encrpytion application.

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.*, 2008.

[2] [Online]. Available: http://sunaytripathi.wordpress.com/2012/07/31/how-does-openflow-sdn-help-virtualizationcloud-part-3-of-3-why-nicira-had-to-do-a-deal/.

[3] "Intel IXP Network Processors," [Online]. Available: http://www.intel.com/p/en_US/embedded/hwsw/hardware/ixp-4xx.

[4] C. Networks, "Octeon Programmers Guide," [Online]. Available: http://university.caviumnetworks.com/downloads/Mini_version_of_Prog_Guide_EDU_July_2010.pdf.

[5] "Netronome Flow Processors," [Online]. Available: http://www.netronome.com/pages/flow-processors/.

[6] homepage., "Open vSwitch," [Online]. Available: http://openvswitch.org/..

[7] J. K. M. S. S. U. A. S. A. K. Syed Akbar Mehdi, "NOX-at-Home Demonstration," in *Demo at 10th GENI Engineering Conference (GEC - 2011)* , Puerto Rico., March 16 – 18, 2011.

[8] S. K. G. L. a. K. S. R. W. J. H. K. S. A. K. Rajesh Narayanan, "Macroflows and Microflows: Enabling Rapid Network Innovation through a Split SDN Data Plane," in *EWSDN*, 2012.

[9] M. B. Anwer, M. Motiwala, M. b. Tariq and N. Feamster, "SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware," in Proceeedings of SIGCOMM 2010.

[10] Cisco, "Cisco's One Platform Kit (onePK)," Cisco, [Online]. Available: http://www.cisco.com/en/US/prod/iosswrel/onepk.html.

[11] "Floodlight OpenFlow Controller," [Online]. Available: http://floodlight.openflowhub.org/.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105--110, July 2008.

[13] "Beacon Controller," [Online]. Available: http://www.openflowhub.org/display/Beacon/.

[14] N. Rajesh, K. Saikrishna, L. Geng, K. aimal, R. Sajjad, J. Wajeeha, K. Hassan and K. Syed Ali, "Macroflows and Microflows: Enabling Rapid Network Innovation through a Split SDN Data Plane," in *EWSDN*, 2012.

[15] Paul Buckley, "HiGig Support on FPGA," [Online]. Available: http://www.electronics-eetimes.com/en/lattice-unveils-first-low-cost-fpga-to-support-broadcom-higig-protocol.html?cmp_id=7&news_id=222904808.

[16] J. Evans, "The Walled Garden Has Won," Techcrunch, [Online]. Available: http://techcrunch.com/2011/03/12/the-walled-garden-has-won/.

[17] J. Yonan. [Online]. Available: http://openvpn.net/papers/BLUG-talk/BLUG-talk.ppt.

[18] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury and A. T. Campbell, "A survey of mobile phone sensing," *Comm. Mag.*, vol. 48, pp. 140--150, sep 2010.

[19] "Ostinato: Packet Generator and Analyzer," [Online]. Available: http://code.google.com/p/ostinato/.

[20] "WireShark," [Online]. Available: http://www.wireshark.org/.

[21] "Nox controller," [Online]. Available: http://noxrepo.org/wp/?page_id=2.

[22] M. Syed Akbar, K. Junaid, S. Mohsin, A. Syed Usman and K. Syed Ali, "NOX-at-Home Demonstration," in *Demo at 10th GENI Engineering Conference (GEC - 2011)*, Puerto Rico., March 16 – 18, 2011.