OFLOPS: An Open Framework for OpenFlow Switch Evaluation

Charalampos Rotsos[1], Nadi Sarrar[2], Steve Uhlig[2], Rob Sherwood[3] *, and Andrew W. Moore[1]

[1] University of Cambridge [2] TU Berlin / T-Labs [3] Big Switch Networks

Abstract. Recent efforts in software-defined networks, such as OpenFlow, give unprecedented access into the forwarding plane of networking equipment. When building a network based on OpenFlow however, one must take into account the performance characteristics of particular OpenFlow switch implementations. In this paper, we present OFLOPS, an open and generic software framework that permits the development of tests for OpenFlow-enabled switches, that measure the capabilities and bottlenecks between the forwarding engine of the switch and the remote control application. OFLOPS combines hardware instrumentation with an extensible software framework.

We use OFLOPS to evaluate current OpenFlow switch implementations and make

We present OFLOPS[2], a tool that enables the rapid development of use-case tests for both hardware and software OpenFlow implementations. We use OFLOPS to test publicly available OpenFlow software implementations as well as several OpenFlow-enabled commercial hardware platforms, and report our findings about their varying performance characteristics. To better understand the behavior of the tested OpenFlow implementations, OFLOPS combines measurements from the OpenFlow control channel with data-plane measurements. To ensure sub-millisecond-level accuracy of the measurements, we bundle the OFLOPS software with specialized hardware in the form of the NetFPGA platform[3]. Note that if the tests do not require millisecond-level accuracy, commodity hardware can be used instead of the NetFPGA [5].

The rest of this paper is structured as follows. We first present the design of the OFLOPS framework in Section 2. We describe the measurement setup in Section 3. We describe our measurements in Section 4. We provide basic experiments that test the flow processing capabilities of the implementations (Section 4.1) as well as the performance and overhead of the OpenFlow communication channel (Section 4.2). We follow with specific tests, targeting the monitoring capabilities of OpenFlow (Section 4.3) as well as interactions between different types of OpenFlow commands (Section 4.4). We conclude in Section 5.

2 OFLOPS framework

Measuring OpenFlow switch implementations is a challenging task in terms of characterization accuracy, noise suppression and precision. Performance characterization is not trivial as most OpenFlow-enabled devices provide rich functionality but do not disclose implementation details. In order to understand the performance impact of an experiment, multiple input measurements must be monitored concurrently. Furthermore, measurement noise minimization can only be achieved through proper design of the measurement platform. Current controller designs, like [11, 3], target production networks and thus are optimized for throughput maximization and programmability, but incur high measurement inaccuracy. Finally, high precision measurements after a point are subject to loss due to unobserved parameters of the measurement host, such as OS scheduling and clock drift.

The OFLOPS design philosophy is to enable seamless interaction with an OpenFlow-enabled device over multiple data channels without introducing significant additional processing delays. The platform provides a unified system that allows developers to control and receive information from multiple control sources: data and control channels as well as SNMP to provide specific switch-state information. For the development of measurement experiments over OFLOPS, the platform provides a rich, event-driven, API that allows developers to handle events programatically in order to implement and measure custom controller functionality. The current version is written predominantly in C. Experiments are compiled as shared libraries and loaded at runtime using a simple configuration language, through which experimental parameters

[2] OFLOPS is under GPL licence and can be downloaded from `http://www.openflow.org/wk/index.php/Oflops`

[3] `http://www.netfpga.org`

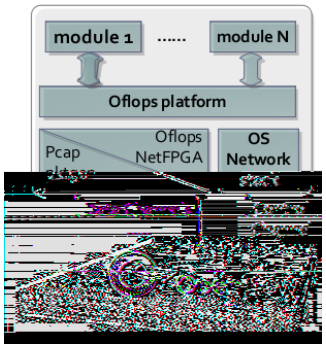


Fig. 1. OFLOPS design schematic

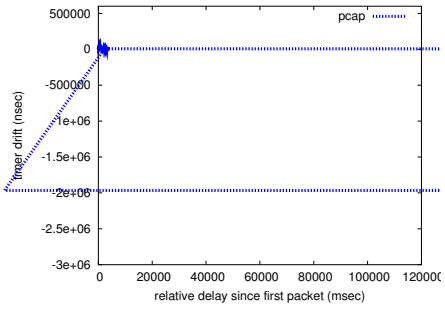


Fig. 2. Evaluating timestamping precision using a DAG card.

can be defined. A schematic of the platform is presented in Figure 1. Details of the OFLOPS programming model can be found in the API manual [1].

The platform is implemented as a multi-threaded application, to take advantage of modern multicore environments. To reduce latency, our design avoids concurrent access controls: we leave any concurrency-control complexity to individual module implementations. OFLOPS consists of the following five threads, each one serving specific type of events:

1. **Data Packet Generation** controls data plane traffic generators.
2. **Data Packet Capture** captures and pushes data plane traffic to modules.
3. **Control Channel** translates OpenFlow packets to control events.
4. **SNMP Channel** performs asynchronous SNMP polling.
5. **Time Manager** manages time events scheduled by measurement modules.

OFLOPS provides the ability to control concurrently multiple data channels to the switch. By embedding the data channel within the platform, it is possible to understand the impact of the measurement scenario on the switching plane. To enable our platform to run on multiple heterogeneous platforms, we have integrated support for multiple packet generation and capturing mechanisms. For the packet generation functionality, OFLOPS supports three mechanisms: user-space, kernel-space through the pktgen module [16], and hardware-accelerated through an extension of the design of the NetFPGA Stanford Packet Generator [8]. For the packet capturing and timestamping, the platform supports both the pcap library and the modified NetFPGA design. Each approach provides different precisions and different impacts upon the measurement platform.

A comparison of the precision of the traffic capturing mechanisms is presented in Figure 2. In this experiment we use a constant rate 100 Mbps probe of small packets for a two minute period. The probe is duplicated, using an optical wiretap with negligible delay, and sent simultaneously to OFLOPS and to a DAG card. In the figure, we plot the differences of the relative timestamp between each OFLOPS timestamping mechanism and the DAG card for each packet. From the figure, we see that the pcap timestamps drift by 6 milliseconds after 2 minutes. On the other hand, the NetFPGA timestamping mechanism has a smaller drift at the level of a few microseconds during the same period.

3 Measurement setup

The number of OpenFlow-enabled devices has slowly increased recently, with switch and router vendors providing experimental OpenFlow support such as prototype and evaluation firmware. At the end of 2009, the OpenFlow protocol specification was released in its first stable version 1.0 [2], the first recommended version implemented by vendors for production systems. Consequently, vendors did proceed on maturing their prototype implementations, offering production-ready OpenFlow-enabled switches today. Using OFLOPS, we evaluate OpenFlow-enabled switches from three different switch vendors. Vendor 1 has production-ready OpenFlow support, whereas vendors 2 and 3 at this point only provide experimental OpenFlow support. The set of selected switches provides a representative but not exhaustive sample of available OpenFlow-enabled top-of-rack-style switching hardware. Details regarding the CPU and the size of the flow table of the switches are provided in Table 1.

OpenFlow is not limited to hardware. The OpenFlow protocol reference is the software switch, OpenVSwitch [17], an important implementation for production environments. Firstly, OpenVSwitch provides a replacement for the poor-performing Linux bridge [7], a crucial functionality for virtualised operating systems. Secondly, several hardware switch vendors use OpenVSwitch as the basis for the development of their own OpenFlow-enabled firmware. Thus, the mature software implementation of the OpenFlow protocol is ported to commercial hardware, making certain implementation bugs less likely to (re)appear. In this paper, we study OpenVSwitch alongside our performance and scalability study of hardware switches. Finally, in our comparison we include the OpenFlow switch design for the NetFPGA platform [15]; a full implementation of the protocol, limited though in capabilities due to hardware platform limitations.

Switch	CPU	Flow table size
Switch1	PowerPC 500MHz	3072 mixed flows
Switch2	PowerPC 666MHz	1500 mixed flows
Switch3	PowerPC 828MHz	2048 mixed flows
OpenVSwitch	Xeon 3.6GHz	1M mixed flows
NetFPGA	DualCore 2.4GHz	32K exact & 100 wildcard

Table 1. OpenFlow switch details.

In order to conduct our measurements, we setup OFLOPS on a dual-core 2.4GHz Xeon server equipped with a NetFPGA card. For all the experiments we utilize the NetFPGA-based packet generating and capturing mechanism. 1Gbps control and data channels are connected directly to the tested switches. We measure the processing delay incurred by the NetFPGA-based hardware design to be a near-constant 900 nsec independent of the probe rate.

4 Evaluation

In this section we present a set of tests performed by OFLOPS to measure the behavior and performance of OpenFlow-enabled devices. These tests target (1) the OpenFlow packet processing actions, (2) the update rate of the OpenFlow flow table along with its

Mod. type	Switch1			OpenVSwitch			Switch2			Switch3			NetFPGA		
	med	sd	loss%	med	sd	loss%	med	sd	loss%	med	sd	loss%	med	sd	loss%
Forward	4	0	0	35	13	0	6	0	0	5	0	0	3	0	0
MAC addr.	4	0	0	35	13	0	302	727	88	-	-	100	3	0	0
IP addr.	3	0	0	36	13	0	302	615	88	-	-	100	3	0	0
IP ToS	3	0	0	36	16	0	6	0	0	-	-	100	3	0	0
L4 port	3	0	0	35	15	0	302	611	88	-	-	100	3	0	0
VLAN pcp	3	0	0	36	20	0	6	0	0	5	0	0	3	0	0
VLAN id	4	0	0	35	17	0	301	610	88	5	0	0	3	0	0
VLAN rem.	4	0	0	35	15	0	335	626	88	5	0	0	3	0	0

Table 2. Time in μs to perform individual packet modifications and packet loss. Processing delay indicates whether the operation is implemented in hardware ($<10\mu$s) or performed by the CPU ($>10\mu$s).

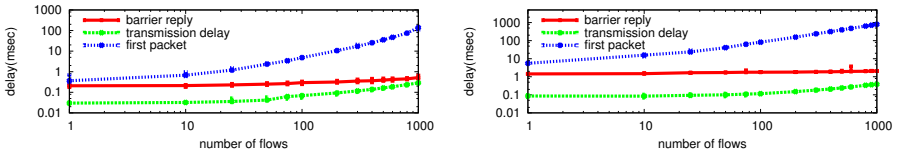
impact on the data plane, (3) the monitoring capabilities provided by OpenFlow, and (4) the impact of interactions between different OpenFlow operations.

4.1 Packet modifications

The OpenFlow specification [2] defines ten packet modification actions which can be applied on incoming packets. Available actions include modification of MAC, IP, and VLAN values, as well as transport-layer fields and flows can contain any combination of them. The left column of Table 2 lists the packet fields that can be modified by an OpenFlow-enabled switch. These actions are used by network devices such as IP routers (e.g., rewriting of source and destination MAC addresses) and NAT (rewriting of IP addresses and ports). Existing network equipment is tailored to perform a subset of these operations, usually in hardware to sustain line rate. On the other hand, how these operations are to be used is yet to be defined for new network primitives and applications, such as network virtualization, mobility support, or flow-based traffic engineering.

To measure the time taken by an OpenFlow implementation to modify a packet field header, we generate from the NetFPGA card UDP packets of 100 bytes at a constant rate of 100Mbps (approx. 125 Kpps). This rate is high enough to give statistically significant results in a short period of time. The flow table is initialized with a flow that applies a specific action on all probe packets and the processing delay is calculated using the transmission and receipt timestamps, provided by the NetFPGA. Evaluating individual packet field modification, Table 2 reports the median difference between the generation and capture timestamp of the measurement probe along with its standard deviation and percent of lost packets.

We observe significant differences in the performance of the hardware switches due in part to the way each handles packet modifications. Switch1, with its production-grade implementation, handles all modifications in hardware; this explains its low packet processing delay between 3 and 4 microseconds. On the other hand, Switch2 and Switch3 each run experimental firmware providing only partial hardware support for OpenFlow actions. Switch2 uses the switch CPU to perform some of the available field modifications, resulting in two orders of magnitude higher packet processing delay and variance. Switch3 follows a different approach: All packets of flows with actions not supported in hardware are silently discarded. The performance of the OpenVSwitch software im-



(a) OpenVSwitch (log-log scale) (b) Switch1 (log-log scale)

Fig. 3. Flow entry insertion delay: as reported using the `barrier` notification and as observed at the data plane.

plementation lies between Switch1 and the other hardware switches. OpenVSwitch fully implements all OpenFlow actions. However, hardware switches outperform OpenVSwitch when the flow actions are supported in hardware.

We conducted a further series of experiments with variable numbers of packet modifications as flow actions. We observed, that the combined processing time of a set of packet modifications is equal to the highest processing time across all individual actions in the set.

4.2 Flow table update rate

The flow table is a central component of an OpenFlow switch and is the equivalent of a Forwarding Information Base (FIB) on routers. Given the importance of FIB updates on commercial routers, e.g., to reduce the impact of control plane dynamics on the data plane, the FIB update processing time of commercial routers provide useful reference points and lower bounds for the time to update a flow entry on an OpenFlow switch. The time to install a new entry on commercial routers has been reported in the range of a few hundreds of microseconds [18].

OpenFlow provides a mechanism to define barriers between sets of commands: the `barrier` command. According to the OpenFlow specification [2], the barrier command is a way to be notified that a set of OpenFlow operations has been completed. Further, the switch has to complete the set of operations issued prior to the barrier before executing any further operation. If the OpenFlow implementations comply with the specification, we expect to receive a barrier notification for a flow modification once the flow table of the switch has been updated, implying that the change can be seen from the data plane.

We check the behavior of the tested OpenFlow implementations, finding variation among them. For OpenVSwitch and Switch1, Figure 3 shows the time to install a set of entries in the flow table. The NetFPGA-based switch results (not reported) are similar to those of Switch1, while Switch2 and Switch3 are not reported as this OpenFlow message is not supported by the firmware. For this experiment, OFLOPS relies on a stream of packets of 100 bytes at a constant rate of 10Mbps that targets the newly installed flows in a round-robin manner. The probe achieves sufficiently low inter-packet periods in order to measure accurately the flow insertion time.

In Figure 3, we show three different times. The first, *barrier notification*, is derived by measuring the time between when the **first insertion command** is sent by the OFLOPS controller and the time the barrier notification is received by the PC. The second, *transmission delay*, is the time between the first and last flow insertion commands

are sent out from the PC running OFLOPS. The third, *first packet*, is the time between the **first insertion command** is issued and a packet has been observed for the last of the (newly) inserted rules. For each configuration, we run the experiment 100 times and Figure 3 shows the median result as well as the 10^{th} and 90^{th} percentiles (variations are small and cannot be easily viewed).

From Figure 3, we observe that even though the *transmission delay* for sending flow insertion commands increases with their number, this time is negligible when compared with data plane measurements (*first packet*). Notably, the *barrier notification* measurements are almost constant, increasing only as the transmission delay increases (difficult to discern on the log-log plot) and, critically, this operation returns before any *first packet* measurement. This implies that the way the *barrier notification* is implemented does not reflect the time when the hardware flow-table has been updated.

In these results we demonstrate how OFLOPS can compute per-flow overheads. We observe that the flow insertion time for Switch1 starts at 1.8ms for a single entry, but converges toward an approximate overhead of 1ms per inserted entry as the number of insertions grows.

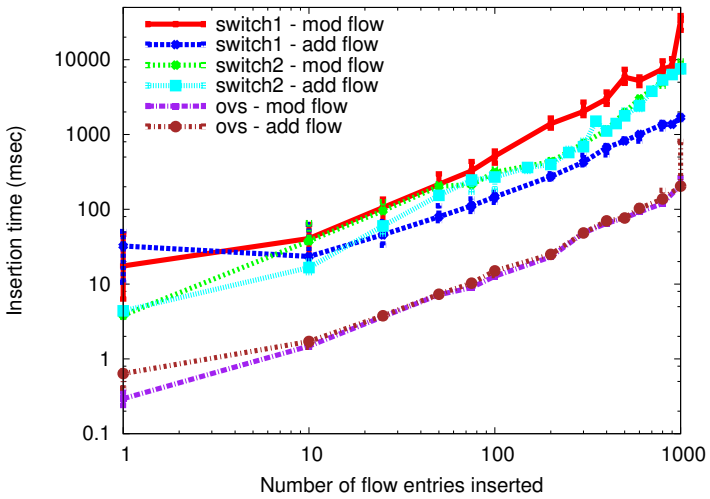


Fig. 4. Delay of flow insertion and flow modification, as observed from the data plane (log-log scale).

Flow insertion types We now distinguish between flow insertions and the modification of existing flows. With OpenFlow, a flow rule may perform exact packet matches or use wild-cards to match a range of values. Figure 4 compares the flow insertion delay as a function of the number of inserted entries. This is done for the insertion of new entries and for the modification of existing entries.

These results show that for software switches that keep all entries in memory, the type of entry or insertion does not make a difference in the flow insertion time. Surprisingly, both Switch1 and Switch2 take more time to modify existing flow entries compared to adding new flow entries. For Switch1, this occurs for more than 10 new entries, while for Switch2 this occurs after a few tens of new entries. After discussing

this issue with the vendor of Switch2, we came to the following conclusion: as the number of TCAM entries increases, updates become more complex as they typically requires re-ordering of existing entries.

Clearly, the results depend both on the entry type and implementation. For example, exact match entries may be handled through a hardware or software hash table. Whereas, wild-carded entries, requiring support for variable length lookup, must be

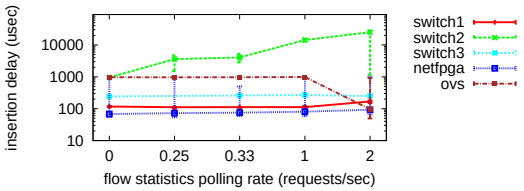


Fig. 6. Delay when updating flow table while the controller polls for statistics.

utilization, answering flow-stats requests as fast as possible (Switch2), or the switch delays responses, avoiding over-loading its CPU (Switch1). Furthermore, for Switch1, we notice that the switch is applying a pacing mechanism on its replies. Specifically, at low polling rates the switch splits its answer across multiple TCP segments: each segment containing statistics for a single flow. As the probing rate increases, the switch will aggregate multiple flows into a single segment. This suggests that independent queuing mechanisms are used for handling flow statistics requests. Finally, neither software nor NetFPGA switches see an impact of the flow-stats rate on their CPU, thanks to their significantly more powerful PC CPUs (Table 1).

4.4 OpenFlow command interaction

An advanced feature of the OpenFlow protocol is its ability to provide applications with, e.g., flow arrival notifications from the network, while simultaneously providing fine-grain control of the forwarding process. This permits applications to adapt in real time to the requirements and load of the network [12, 21]. Under certain OpenFlow usage scenarios, e.g., the simultaneous querying of traffic statistics and modification of the flow table, understanding the behavior of the data and control plane of OpenFlow switches is difficult without advanced measurement instrumentation such as the one provided by OFLOPS.

Through this scenario, we extend Section 4.2 to show how the mechanisms of traffic statistics extraction and table manipulation may interact. Specifically, we initialize the flow table with 1024 exact match flows and measure the delay to update a subset of 100 flows. Simultaneously, the measurement module polls the switch for full table statistics at a constant rate. The experiment uses a constant rate 10Mbps packet probe to monitor the data path, and polls every 10 seconds for SNMP CPU values.

In this experiment, we control the probing rate for the flow statistics extraction mechanism, and we plot the time necessary for the modified flows to become active in the flow table. For each probing rate, we repeat the experiment 50 times, plotting the median, 10^{th} and 90^{th} percentile. In Figure 6 we can see that, for lower polling rates, implementations have a near-constant insertion delay comparable to the results of Section 4.2. For higher probing rates on the other hand, Switch1 and Switch3 do not differ much in their behavior. In contrast, Switch2 exhibits a noteworthy increase in the insertion delay explained by the CPU utilization increase incurred by the flow statistics polling (Figure 5(b)). Finally, OpenVSwitch exhibits a marginal decrease in the median insertion delay and at the same time an increase in its variance. We believe this behavior is caused by interactions with the OS scheduling mechanism: the constant polling causes frequent interrupts for the user-space daemon of the switch, which leads to a batched handling of requests.

5 Summary and Conclusions

We presented, OFLOPS, a tool that tests the capabilities and performance of OpenFlow-enabled software and hardware switches. OFLOPS combines advanced hardware instrumentation, for accuracy and performance, and provides an extensible software framework. We use OFLOPS to evaluate five different OpenFlow switch implementations, in terms of OpenFlow protocol support as well as performance.

We identify considerable variation among the tested OpenFlow implementations. We take advantage of the ability of OFLOPS for data plane measurements to quantify accurately how fast switches process and apply OpenFlow commands. For example, we found that the barrier reply message is not correctly implemented, making it difficult to predict when flow operations will be seen by the data plane. Finally, we found that the monitoring capabilities of existing hardware switches have limitations in their ability to sustain high rates of requests. Further, at high rates, monitoring operations impact other OpenFlow commands.

We hope that the use of OFLOPS will trigger improvements in the OpenFlow protocol as well as its implementations by various vendors.

References

1. OFLOPS. `http://www.openflow.org/wk/index.php/Oflops`.
2. Openflow switch specification (version 1.0.0). `www.openflow.org/documents/openflow-spec-v1.0.0.pdf`, December 2009.
3. The snac openflow controller, 2010. `http://www.openflow.org/wp/snac/`.
4. Agilent. N2X router tester. `http://advanced.comms.agilent.com/n2x/`.
5. P. Arlos and M. Fiedler. A method to estimate the timestamp accuracy of measurement hardware and software tools. In *PAM*, 2007.
6. G. Balestra, S. Luciano, M. Pizzonia, and S. Vissicchio. Leveraging router programmability for traffic matrix computation. In *Proc. of PRESTO workshop*, 2010.
7. A. Bianco, R. Birke, L. Giraudo, and M. Palacin. Openflow switching: Data plane performance. In *IEEE ICC*, may 2010.
8. G. Covington, G. Gibb, J. Lockwood, and N. Mckeown. A packet generator on the NetFPGA platform. In *FCCM 2009*.
9. A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow:scaling flow management for high-performance networks. In *ACM SIGCOMM*, 2011.
10. D. A. Freedman, T. Marian, J. H. Lee, K. Birman, H. Weatherspoon, and C. Xu. Exact temporal characterization of 10 gbps optical wide-area network. In *IMC 2010*.
11. 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.*, July 2008.
12. N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. In *ACM SIGCOMM Demo*, August 2009.
13. Ixia. Interfaces. `http://www.ixiacom.com/`.
14. L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *Proc. of the USENIX HotICE workshop*, 2011.
15. J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. In *ANCS*, 2008.
16. R. Olsson. pktgen the linux packet generator. In *Proceedings of Linux symposium*, 2005.

17. J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby. Virtualizing the network forwarding plane. In *DC-CAVES*, 2010.
18. A. Shaikh and A. Greenberg. Experience in black-box ospf measurement. In *ACM IMC*, 2001.
19. R. Sherwood, G. Gibb, K. Yapa, M. Cassado, G. Appenzeller, N. McKeown, and G. Parulkar. Can the production network be the test-bed? In *OSDI*, 2010.
20. A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: traffic matrix estimator for openflow networks. In *PAM*, 2010.
21. K.-K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown. The stanford openroads deployment. In *Proceedings of ACM WINTECH*, 2009.
22. M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *ACM SIGCOMM*, August 2010.