 The definitive version of this paper has been published in 1st European Workshop on Software Defined Networks (EWSDN 2012), 2012, 10.1109\/ewsdn.2012.15.

A Flexible OpenFlow-Controller Benchmark

Michael Jarschel, Frank Lehrieder, Zsolt Magyari, Rastin Pries
University of Würzburg, Institute of Computer Science, Würzburg, Germany.
Email: {michael.jarschel,lehrieder,pries}@informatik.uni-wuerzburg.de, zsolt.magyarizsolt@googlemail.com

***Abstract*—Software defined networking (SDN) promises a way to more flexible networks that can adapt to changing demands. At the same time these networks should also benefit from simpler management mechanisms. This is achieved by moving the network control out of the forwarding devices to purpose-tailored software-applications on top of a "networking operating system". Currently, the most notable representative of this approach is OpenFlow. In the OpenFlow architecture the operating system is represented by the OpenFlow controller. As the key component of the OpenFlow ecosystem, the behavior and performance of the controller are significant for the entire network. Therefore, it is important to understand these influence factors, when planning an OpenFlow-based SDN deployment. In this work, we introduce a tool to help achieving just that - a flexible OpenFlow controller benchmark. The benchmark creates a set of message-generating virtual switches, which can be configured independently from each other to emulate a certain scenario and also keep their own statistics. This way a granular controller performance analysis is possible.**

***Keywords*-OpenFlow, controller, performance evaluation**

I. Introduction

OpenFlow [1] has emerged as an important enabler for software defined networking. As the key component in the OpenFlow architecture, OpenFlow controllers are often marketed as "networking operating system" in a software defined network. However, this designation is slightly misleading. While the OpenFlow controller certainly fills the role of an operating system bridging the gap between physical hardware and applications, many controllers lack the stability and performance we would expect a modern operating system to have in the computing domain. OpenFlow controllers can not be configured, but have to be programmed, which makes them more akin to operational frameworks than an actual operating system. Since the OpenFlow standard does not dictate how a controller should be implemented or even which elements it should possess beyond the OpenFlow secure channel, a variety of different implementations has been developed, each with its own behavior and performance characteristics. These differences make specific controllers better suited for certain scenarios than others. To choose an implementation over another and to analyze the system behavior for a particular deployment, these differences have to be understood. In this paper we introduce a flexible OpenFlow controller benchmark as a tool to obtain that insight. Unlike conventional benchmarks that focus on overall throughput and/or latency, our benchmark allows the emulation of scenarios and topologies and can evaluate the controller performance on a per-switch basis. This way a more detailed analysis of controller performance bottlenecks as well as obscure behavior is possible. The purpose of this contribution is on the one hand to show the features of our tool, which we will publish as open-source software, and on the other hand to underline the necessity of tools like this for analyzing and understanding controller and, as a consequence, OpenFlow network performance.

The remainder of this paper is structured as follows. In Section II we discuss the work related to OpenFlow benchmarking. We introduce our benchmark architecture in Section III. Subsequently, we compare the performance of our benchmark to that of the standard OpenFlow controller benchmark "Cbench" [2] in Section IV. We proceed by discussing first benchmarking results in Section V. Finally, we draw our conclusions in Section VI.

II. Related Work

Current deployments of OpenFlow mostly rely on conventional switches as forwarding units. As these are usually not designed to function as flow switches, they are often performance bottlenecks and thus the research focus in terms of performance so far lay on the forwarding plane. In [3], Bianco et al. measure the OpenFlow switching performance for different types of rules and packet sizes. A similar approach is taken by Sünnen in his student thesis [4]. The author analyzes the performance impact of the flow table in various scenarios. Curtis et al. [5] propose changes to the OpenFlow protocol as they discovered inherent performance bottlenecks with regard to CPU load in current OpenFlow switch implementations.

In Pries et al. [6], we evaluated the usability of OpenFlow in data centers. We discovered that setting flow rules reactively leads to an unacceptable performance when only eight switches are handled by one controller. In [7] we introduced an analytical model to predict the OpenFlow system performance. We aim to improve this model through insights gained from benchmarking results.

With OFlops [8] a framework for performance analysis of OpenFlow switches exists. However, on the controller side only a relatively simple benchmark exists with Cbench [2]. Cbench was first developed by Robert Sherwood and has since become the standard evaluation tool for controller

performance. In [9] Tootoonchian et al. use Cbench to highlight possible controller performance improvements. Still, little can be derived from the results in terms of controller behavior. Cbench is single-threaded, i.e. multiple instances have to be started to utilize multiple CPUs. It also only uses one controller connection for all emulated switches. Aggregated statistics are gathered for all switches but not for each switch individually. As a result, it is for example not possible to tell whether all controller responses are for a single switch and the others receive nothing, or whether the controller capacity is shared fairly among the switches. Our benchmark addresses these issues to obtain more granular results.

III. Benchmark Architecture

In this section we present our OFCBenchmark tool. First, we explain the design goals that guided our development process. Then, we present the architecture of the software and describe the implementation.

A. Design Goals

The architecture and the implementation of OFCBenchmark are guided by the following main design goals.

- *Scalability:* The software should be designed in a way so that multiple instances can run in a coordinated way on different CPU cores, CPUs, and hosts. This achieves that the load generated to test an OpenFlow controller is not limited by a single core of the CPU or the memory of the machine that runs the software.
- *Ability to provide detailed performance statistics:* Our benchmarking software should provide performance metrics such as round trip times, sent or received packets per second, or the number of outstanding packets, in time series and on a per switch basis. This feature permits to investigate whether an OpenFlow controller treats switches differently or changes its behavior over time.
- *Modularity:* The controller development process progresses rapidly and therefore, the benchmark should be adaptable to new scenarios, which is easier in a modularized software. In addition, measurements of further performance metrics and further parameters to control the load generation should be easy to add to the benchmarking software.

B. Architecture Design

The architecture of our OpenFlow controller benchmark consists of three main components - the OFCBenchmark Control Center (OCC), the OFCBenchmark Client (OC), and the Virtual Switch (VS). The OFCBenchmark uses a distributed approach, i.e. the benchmark can be spread over multiple hosts. Each of these hosts runs an instance of the OC. The OC itself is already a full benchmarking system.

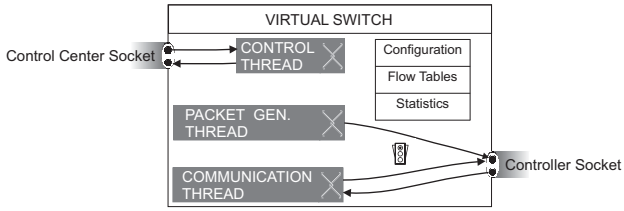


Figure 1. Structure of a Virtual Switch.

It executes the performance tests using the configured number of VS objects. However, it is limited in scale as the number of VSs is restricted by the amount of memory and computing capacity of a single host. In the distributed mode multiple OCs connect to the OCC, which then controls the experiment.

The key component of the OFCBenchmark is the virtual switch. Figure 1 shows a schematic illustration of the VS structure. The virtual switch holds a simplified flow table to be able to respond to controller requests. It also has a statistics store where the benchmarking results are kept and updated. Furthermore, every virtual switch has two socket connections and three threads encapsulated in the virtual switch object. The connections serve as communication channels to the OCC and the OpenFlow Controller and are managed using the threads. This allows us to treat a VS as a true individual entity, which gives us the desired modularity, scalability, and the ability to provide detailed performance statistics.

C. Implementation

The OC and its VSs are written in C++ using the Boost library for thread-handling. Experiments can either be configured directly via the OCC communications channel in distributed mode or through a configuration file in standalone mode. Configuration options include number of switches, per switch packet-inter-transmission times, packet sizes, as well as the option to specify a pcap file containing OpenFlow messages for each switch to play-out. Furthermore, the OC allows the creation of a switch topology specified in a separate configuration file. This is achieved by allowing the virtual switches to seemingly forward controller generated LLDP or OFLDP packets and sending them back to the controller as Packet-In messages according to the configured topology.

At creation time the VS reads the same configuration file as the OC and connects itself to the OCC waiting for further instructions or requests. In Figure 1 we can see the control thread connected with the OCC through a socket. This thread executes the commands of the OCC in the VS. If the connection to the OCC is not configured in the configuration file, the switches are operated in stand-alone mode by the OC. The two remaining threads are using the

same TCP socket and are the workers of the virtual switch.

The choice of TCP as transmission protocol reflects the OpenFlow specification. The communication-thread is responsible for handling the communication with the currently benchmarked controller using the OpenFlow protocol in version 1.0. It performs the OpenFlow handshake process and answers other controller requests. The packet-generator thread creates and sends Packet-In messages to the controller for benchmarking. The time between two sent packets can be configured. By default the time is set to zero, which results in as many packets being sent as the TCP stream allows. The Packet-In messages contain the packet header of the first packet of a new IP flow the controller has not yet encountered. Every Packet-In message is identifiable through its buffer-id. The controller responds to those packets with Packet-Out and/or FlowMod messages using the same buffer-id to identify the corresponding packet. Receiving the response, the communication thread parses the id, calculates the round trip time for this request and updates the statistics. As both threads are using the same socket, a semaphore was included to coordinate the output and avoid data corruption. This is a mutual exclusion that prevents the usage of socket output from different threads at the same time. Before sending a datagram the "user" of the socket output has to lock the semaphore and release it after transmission. To be able to use multiple CPUs and thus to keep accurate statistics for the round trip time, the virtual switches use blocking I/O, i.e. a thread is only "woken" by the operating system once a packet arrives. This way we avoid having to check frequently whether a packet has arrived using a single thread.

The OCC is a graphical user interface written in Delphi, so experimenters can see the current configuration at a glance and modify the test settings according to their requirements.

IV. Comparison with Cbench

To verify the results of our benchmark we run a comparative test with Cbench. The test was run on a testbed consisting of two PCs directly connected through a 100 Mbps Link. Both systems share the same hardware and software configuration with a Pentium IV 3.4GHz CPU, 1 GB RAM, and Ubuntu 10.04 as operating system. One PC runs the controller – in this case Nox Classic [10] as learning switch – and the other PC runs the benchmarking tools. The benchmark is set not to introduce artificial delay between packet departures. We compare our benchmark to two versions of Cbench – the current repository version and the original version used for reference.

Figure 2 shows the achieved throughput in packets per second with respect to the number of connected virtual switches. The error-bars attached to the graphs give the 95 percent confidence intervals, which were obtained through five repetitions of each test.

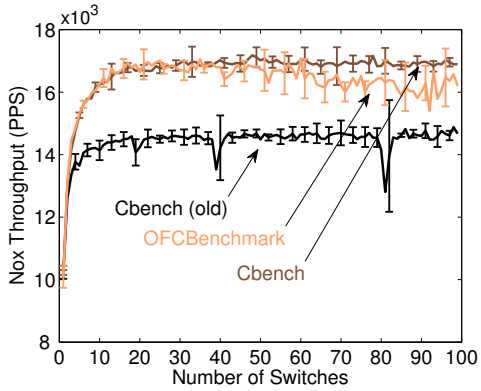


Figure 2. Comparison of our Benchmark with Cbench using the Nox-Classic Controller.

All curves increase from about 10,000 pps with just one virtual switch until they reach a stable level of saturation with about 15 connected switches. The legacy Cbench version reaches its saturation at about 14,300 pps. The current Cbench as well as our OFCBenchmark achieve a higher throughput and reach their saturation at about 16,900 pps. From about 50 connected switches we see a slight decrease to about 16,000 pps in throughput and larger confidence intervals for our benchmark. This is likely due to the larger overhead caused by managing and keeping statistics for each virtual switch independently. However, the difference to Cbench is still quite small and partially still within the confidence intervals. Therefore, we can assume that our benchmark produces comparable results to the Cbench tool.

V. Benchmarking Results

In this section we discuss some initial results we have obtained with our benchmark. These results are produced with a software in development. This is neither an exhaustive and/or representative comparison between the benchmarked controllers, nor can or should a general statement about the quality of the controllers be derived from this simple scenario. This is intended to showcase the features of our benchmark and a discussion of the results and their consequences. The testbed used here is identical to the one described in the previous section. The controllers measured are Nox Classic [10] ("Zaku" release), Floodlight [11] (version 0.82), and Maestro [12] (version 0.2). These controllers were chosen arbitrarily with the only requirement being that they are freely available. All controllers were set to use their respective learning switch applications. As Nox Classic has no multi-threading, the other controllers were also limited to a single thread to obtain comparable results. Nagle's algorithm was deactivated on all test systems to avoid the influence of artificial TCP buffering. All tests were repeated 5 times to obtain the confidence intervals shown in the figures.

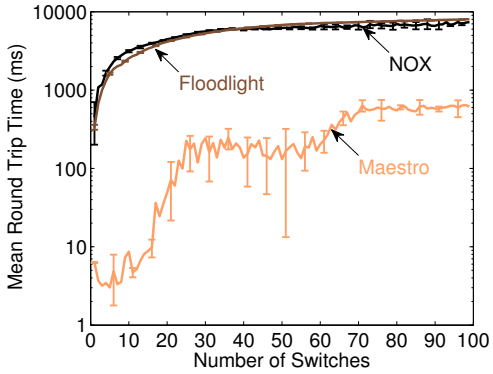


Figure 3. Mean Response Time for an OpenFlow Packet-In Message.

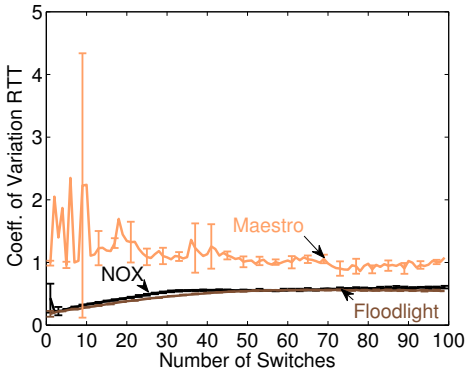


Figure 4. Coefficient of Variation of the Response Time for an OpenFlow Packet-In Message.

A. Mean Round Trip Time

The first feature test is for the round trip time (RTT), i.e. the interval from the moment a Packet-In message is dispatched from the virtual switch to the controller until the corresponding Packet-Out or FlowMod message is received by the switch. This test can also be performed with Cbench. Our benchmark extends this feature by allowing to obtain these statistics for each switch individually and as a time series as well.

Figure 3 illustrates the mean round trip time in milliseconds for different numbers of simultaneously connected switches to the controller. Note that the y-axis is scaled logarithmically. We observe that the Floodlight and Nox Classic controller behave in a similar way. The response time of these controllers increases rapidly from about 200 ms for one switch until the value stabilizes at about 6 seconds for 30 switches. Both controllers are obviously under heavy load at this point due to the relatively weak hardware. However, the RTT for Floodlight continues to increase up to about 8 seconds for 80 switches. Maestro behaves differently. For this scenario Maestro starts at a RTT of only about 6 ms – two orders of magnitude faster than the other controllers. With an increasing number of switches the RTT increases steadily, but far slower compared to the others, until for 100 switches a RTT of just under 1 second is reached. The largest increase in RTT can be observed between 40 and 50 connected switches. We suspect this behavior is the result of a different processing strategy for Maestro that is advantageous in this scenario as we will see indicated by the results shown in Figure 5.

However, the average RTT of all switches and over the whole experiment duration is neither sufficient to judge whether this value changed over time nor whether some switches experienced larger or smaller RTTs, i.e., whether some switches received a preferred treatment. To answer such questions, our tool provides time series of the RTTs on a per switch basis. As an example evaluation of this data, we calculate the average RTT over time for every switch and analyze the variability of this value among the different switches by showing the coefficient of variation (c_V) of these values, cf. Figure 4.

As suggested by the small confidence intervals in Figure 3, the c_V for the RTT is small for all controllers. For Nox Classic and Floodlight we see an increase from about 0.2 with one switch to a stable value of about 0.5 at 30 switches for Nox Classic and 50 switches for Floodlight. Initially Maestro displays a higher c_V at about one with outliers up to 3 and large confidence intervals. This can be explained with the much smaller mean RTT, as small derivations from small mean values have a far larger impact on the c_V than small derivations from larger ones. Mirroring the observations from Figure 4, the c_V decreases between 40 and 50 connected switches to a value of about 0.3 – below that of the other two controllers.

B. Send- and Response Rates

Apart from determining the throughput and latency of an OpenFlow controller, it might also be interesting or even important to look at the rate it accepts packets. This can provide insights into rate control mechanisms and/or polling strategies of the controllers. Therefore, we have included this feature in our benchmark.

In Figure 5 the result for the number of packets per second (pps) sent from the switches to the controller through the OpenFlow secure channel is given. As our benchmark uses TCP to send the Packet-In messages to the controller instead of writing the packets raw on the wire, the send rate is determined by the TCP connection. We observe that the packet send rate for the Floodlight controller does not increase significantly with the number of switches. It starts at a rate of about 10,000 pps for one switch and increases to about 38,000 pps. For Nox Classic the increase is slightly steeper, but stalls at about 70,000 ps for 70 switches. However, for Maestro we see a far higher increase in packet send rate with the number of switches. It increases linearly from about 5000 pps to about 140,000 pps for 35

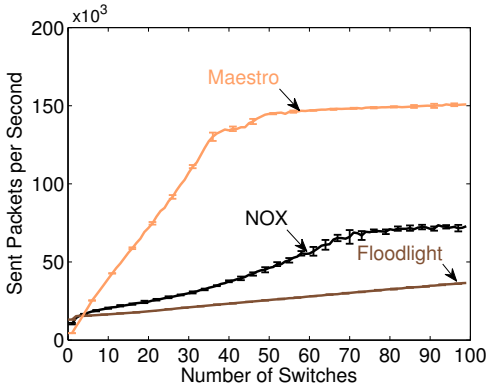


Figure 5. Virtual Switch Packet-In Send-Rate.

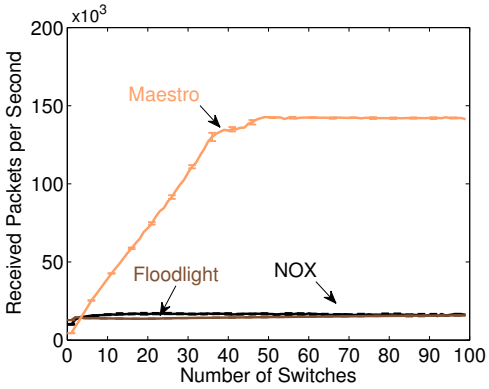


Figure 6. Virtual Switch Packet-Out Reception-Rate.

switches and then continues to increase to 150,000 pps for 50 switches. This suggests the implementation of a rate-control mechanism for Nox Classic and Floodlight, whereas Maestro accepts packets in a best effort manner.

The complement to the send rate is given in Figure 6 – the packet reception rate. It describes the number of responses the switches receive from the controller per second. The figure shows that the reception rate for all controllers is similar to the send rate of the switches shown in the previous figure. However, we do not see an increase in reception rate with an increasing number of switches for Floodlight and Nox as we observed for the rate of sent packets in Figure 5. The reception rate is basically stable at about 10,000 pps. The curve for Maestro displays a steep rate increase identical to the corresponding send rate. The initial rate of about 5000 pps for one switch steadily grows up to about 135,000 pps for 35 switches. As before we see the following flatter rate growth up to about 145,000 pps for 50 switches, where it remains stable. This means for Meastro there is a discrepancy of about 5000 pps between send and reception rate at this point. We call this discrepancy the ”outstanding packets”, i.e. the number of unanswered Packet-In messages by the controller. We take a look at these in the following results using the per switch analysis option of our benchmark.

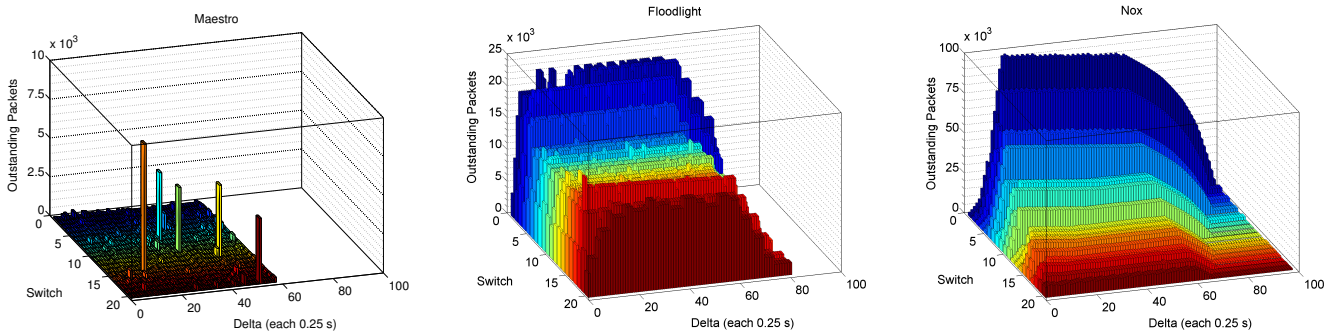
C. Outstanding Packets

Figure 7 shows the evolution of the number of outstanding packets per virtual switch for a test run with 20 connected switches over time for a test run of 15 seconds. The values are presented for time intervals of 0.25 seconds and the time axis in the plots shows the interval number instead of the time value.

Figure 7(a) gives the number of outstanding packets for the Maestro controller. As we can see from the comparison of Figures 5 and 6, the number of outstanding packets is very small for 20 switches. All packets have been processed shortly after the 15-second sending period is over. However, we see some ”spikes” in the graph, i.e. samples with a large number of outstanding packets from about 2000 to 6000 packets.

The number of outstanding packets for the Floodlight controller is given in Figure 7(b). Floodlight shows a different behavior compared to Maestro. All switches have quite a large number of outstanding packets. Switches 1-4 show a particularly high number of 15,000-20,000 outstanding packets and maintain this level over the course of the experiment. The remaining switches hold a level of about 10,000 outstanding packets. This is interesting for two reasons. For one the level of outstanding packets remains constant, i.e. there is no overload in the system. This suggests the presence of a packet buffer in the Floodlight controller. Second, while most switches are treated equally, the first four switches seem to have a larger number of buffered packets. Before the experiment, the virtual switches are connected sequentially to the controller. Therefore, it appears that the order in which the switches are connected has an influence on the buffer size. After the end of the sending period, the buffer is gradually processed as can be seen from the decrease in outstanding packets after 60 samples. It takes an additional 5 seconds after the end of the 15-second experiment until all packets have been answered.

With Nox Classic the influence of the connection order on the number of outstanding packets per switch is even more significant, cf. Figure 7(c). Apparently, Nox Classic does not treat switches equally. The first connected switch experiences a build-up of up to 100,000 outstanding packet. For each subsequent switch the average number of outstanding packets is slightly reduced. The 20th and last connected switch only experiences 10,000-15,000 outstanding packets on average. As a result, all packets of the later connected switches have finished processing only 1-2 seconds after the sending period ends, whereas for the first connected switch the processing takes an additional 10 seconds to complete. In a real network a behavior like this would lead to unfairness. Devices attached to one of the first



(a) Maestro. (b) Floodlight. (c) Nox.

Figure 7. Outstanding Packets per Switch.

switches would experience far larger flow setup times. Using only aggregates and mean values, we would not be able to determine the cause of the issue. While the obtained results may not be 100 percent accurate as the software is still in development, the fact that the results are repeatable and differ between controllers on the same test systems highlights that there are indeed notable differences between controller behaviors. This circumstance and its consequences should be investigated and this is what our approach is aimed at.

VI. Conclusion

Given the importance of the OpenFlow controller for the software defined network it directs, it is key to understand the performance and behavior of this important software component for experimenters as well as for operators of productive networks. In this paper we introduce our approach to a flexible and granular benchmarking system for OpenFlow controllers to create this insight and understanding. We show that the results for conventional throughput tests are comparable to the results of the current reference benchmark Cbench and that it is possible to run the benchmark on conventional hardware. Our benchmark results, especially those for outstanding packets, underline the importance of a more granular view on the system to detect performance bottlenecks and similar issues that can not be grasped from an aggregated perspective. Therefore, we will continue to develop our benchmarking system to generate representative results and an eventual for a public release. We aim to further investigate our finds by studying controller code and gain more insight into the existing OpenFlow controllers using the benchmark for a broader study in the future.

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," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, 2008.

[2] R. Sherwood and K.-K. Yap, "Cbench Controller Benchmarker," http://www.openflowswitch.org/wk/index.php/Oflops, 2011, last accessed on 2012.06.29.

[3] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "OpenFlow Switching: Data Plane Performance," in *IEEE ICC*, Cape Town, South Africa, May 2010.

[4] D. Sünnen, "Performance evalulation of openflow switches," Semester Thesis at the Department of Information Technology and Electrical Engineering, ETH Zürich, February 2011.

[5] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, Toronto, Canada, 2011.

[6] R. Pries, M. Jarschel, and S. Goll, "On the Usability of OpenFlow in Data Center Environments," in *Workshop on Clouds, Networks and Data Centers collocated with the IEEE International Conference on Communications (ICC 2012)*, Ottawa, Canada, Jun. 2012.

[7] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and Performance Evaluation of an OpenFlow Architecture," in *23rd International Teletraffic Congress (ITC 2011)*, San Francisco, CA, USA, Sep. 2011.

[8] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation," in *Passive and Active Measurement Conference (PAM 2012)*, Vienna, AT, March 2012.

[9] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. USENIX Association, 2012.

[10] "Nox Classic," http://www.noxrepo.org/nox/nox-classic-repo/, last accessed on 2012.06.29.

[11] "Floodlight," http://floodlight.openflowhub.org/, last accessed on 2012.06.29.

[12] "Maestro," http://code.google.com/p/maestro-platform/, last accessed on 2012.06.29.