An Architectural Evaluation of SDN Controllers

Syed Abdullah Shah, Jannet Faiz, Maham Farooq, Aamir Shafi, Syed Akbar Mehdi
School of EECS, National University of Sciences and Technology (NUST), Pakistan

***Abstract*—With the recent interest in Software Defined Networking, many OpenFlow controllers have been released for research and commercial use. However, little public knowledge exists about the architectural choices that allow one controller to outperform another in production environments. In this paper, we aim to identify key performance bottlenecks and good architectural choices for designing OpenFlow-based SDN controllers. With this aim in mind, we evaluate the performances of four prominent open-source OpenFlow controllers: NOX [1], Beacon [2], Maestro [3] and Floodlight [4]. Since these controllers support multi-threading, we deploy them on shared memory multicore machines and benchmark their key architectural components under different metrics including thread scalability, switch scalability and latency in a custom cluster testbed. Our results lead to important architectural guidelines that can be used to improve the scalability of existing controllers or to design new ones. We follow these guidelines to implement an OpenFlow controller which outperforms existing controllers on assorted scalability metrics.**

I. INTRODUCTION

The recent commercial interest in SDNs has led to the development of many OpenFlow-based control plane solutions [1]–[4]. While recent studies have investigated the performances of OpenFlow control and data planes [5], [6], the focus thus far has been on identifying *which* controller performs better than the others. We advocate an alternative approach of revealing *why* the performance discrepancies exist between the controllers and, using this knowledge, to determine *how* one can optimize a controller for a certain set of performance metrics. Our evaluation approach requires a thorough understanding of the rationales and implications of the design choices made by existing controllers because, unlike prior studies, our goal is to ascertain key architectural guidelines that can be used to improve the performance of an existing and new controllers.[1]

With the above motivation, in this paper we benchmark the performances of four prominent, publically-available OpenFlow controllers. Since existing controllers utilize multiple threads to enhance performance, we develop a performance evaluation testbed in a cluster environment with multi-core controller machines. In this testbed, we stress different aspects of controller design including network I/O efficiency and scalability.

The main contributions of this paper are:

[1]Indeed, our results show that different controller architectures yield performance improvements for different metrics.

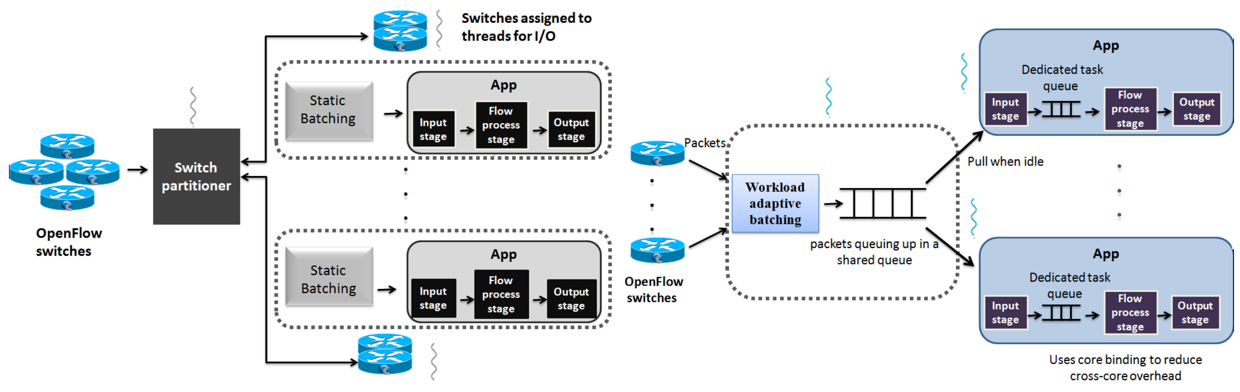
- We identify and compare the architectures of four prominent OpenFlow controllers (NOX, Beacon, Maestro and Floodlight).
- We provide a detailed evaluation of the network I/O and computational efficiencies of these controllers with varying threads and switches.
- We use the results of the evaluation to identify promising architectural guidelines which can be used to improve existing or design new Openflow control planes.
- We use these guidelines to implement a controller which provides improvements of 11.5%, 11.6% and 2.7% (over the best existing controllers) in throughput, latency and switch scalability, respectively.

II. A High-Level Review of Controller Architectures

We start off by comparing the internal architectures of the four open-source OpenFlow controllers being considered in this work. We observe that the key architectural features that differ between these controllers are:
Multicore Support: in terms of the number of main (if any) and worker threads started by the multithreaded controller running on a system with P cores;
Switch Partitioning: defined in terms of the distribution and allocation of connected OpenFlow switches to worker threads running in the controller;
Packet Batching: a technique – used by all OpenFlow controllers – where multiple bytes are read from or written to the underlying network using a socket buffer; this feature helps prevent the overhead of a socket (read/write) system call for a large number of pending requests, thereby improving the overall throughput of the controller;
Task Batching: a strategy used to allocate already-received packets to the worker threads for processing, hence directly impacting the latency of the controller.

These architectural choices are shown pictorially in Figure 1. The first design [Figure 1(a)], uses static switch partitioning and static batching and is employed by Beacon [2], NOX-MT [1] and Floodlight [4]. A switch partitioner module – that runs inside the main thread – listens for incoming switch connections and partitions them amongst the worker threads. Once a switch has been assigned to a particular thread, it is responsible for all the subsequent I/O from that switch.

The second design [Figure 1(b)], uses shared-queue, adaptive-batching design and is employed by Maestro [3]. The main thread is responsible for listening to new switch connections and storing incoming packets in a shared-


Switches assigned to threads for I/O
Static Batching
App
Input stage
Flow process stage
Output stage
OpenFlow switches
Switch partitioner
Packets
Workload adaptive batching
packets queuing up in a shared queue
Pull when idle
Dedicated task queue
Uses core binding to reduce cross-core overhead

(a) Design 1: static partitioning, static batching

(b) Design 2: shared queue, adaptive batching

Fig. 1. General Architectural designs employed by OpenFlow controllers

queue by employing packet batching technique. Once packets have been stored in the shared-queue, worker threads can pull these packets and perform packet processing.

We now discuss how different controllers leverage the above designs and the subsequent impact of these design choices on the controllers' performances.

A. *Maestro*

Maestro is a multi-threaded OpenFlow controller whose architecture is identical to the design shown in Figure 1(b). OpenFlow raw packets are received from the sockets by the main thread and put in the shared raw-packet queue.[2] Packet batching is used but the number of bytes to be read is not static and depends on the present workload (workload adaptive batching). Maestro is the only publicly-available controller which uses task batching so that worker threads pull a batch of tasks to process multiple flow-requests in a single execution. Output batching technique is used to send packets out in which packets belonging to the same destination are grouped together and sent using a single socket system call. Using the workload-adaptive batching technique, Maestro maintains a balance between its throughput and latency performances.

B. *NOX-MT*

NOX is a C++ OpenFlow controller with a Python interface for development purposes. Earlier versions of the controller employed a cooperative threading model which had performance bottlenecks. As a consequence, a multi-threaded architectural design similar to Figure 1(a) was introduced and named NOX-MT. NOX-MT uses the `Boost::Asio` [8] libraries for network and low-level I/O programming. As shown in Figure 1(a), `Boost::Asio` acts as a switch partitioner and is responsible for distributing the connecting OpenFlow switches to worker threads statically. Static packet batching technique is used to reduce frequent `read` system calls. The incoming packets are processed one-by-one (no task batching) and then batched together in case of high control traffic before they are sent out. Static input batching helps NOX-MT in achieving a high throughput performance but affects its latency.

[2]A recent technical report [7] discusses alternate design techniques instead of using shared-queue design.

C. *FloodLight*

Floodlight is a Java-based OpenFlow controller forked from Beacon. It follows the architectural design given in Figure 1(a) and uses Java Netty libraries for handling sockets and employing multithreading in their architecture. A user can configure FloodLight to spawn $n \times P$ threads, where P is the number of available cores and n is user-defined; by default $n = 2$. Floodlight uses oversubscription of threads to ensure that the cores are not underutilized. Static switch partitioning technique is used where a fixed number of switches is assigned by Netty (switch partitioner) to each thread. Static packet batching is employed by using a fixed size read buffer. Per-packet processing is done and then output batching is performed to reduce the overhead of system calls for each individual packet. Floodlight uses static input batching and switch partitioning in its architectural design.

D. *Beacon*

Beacon is a Java-based OpenFlow controller which employs a multi-threaded architectural design similar to that of Figure 1(a). If n number of threads are configured to be run, the controller spawns $n + 1$ threads, where the additional thread is responsible for listening to incoming switch connections and partitioning them among the worker threads. Beacon uses a static approach in which a fixed number of switches are assigned to a worker thread. Worker threads use static packet batching to serve the requests from the connected switches. Once the packets are processed and ready to be sent, Beacon in its default mode uses write coalescing and allows only one write per I/O select loop to reduce the overhead of socket system calls for each individual OpenFlow message. Alternatively, an immediate mode can be enabled in which the controller attempts a socket write for every outgoing OpenFlow

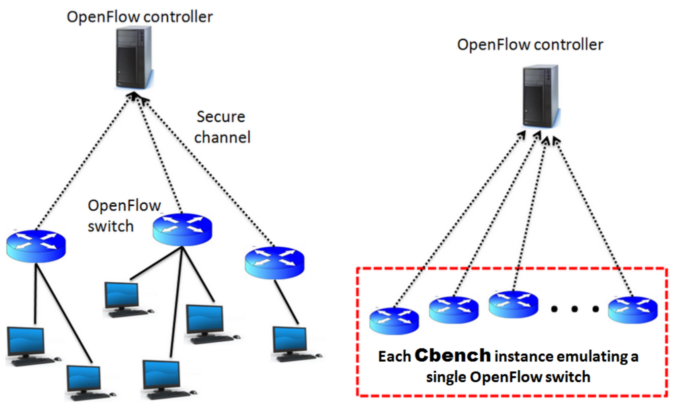


Fig. 2. `Cbench` emulation.

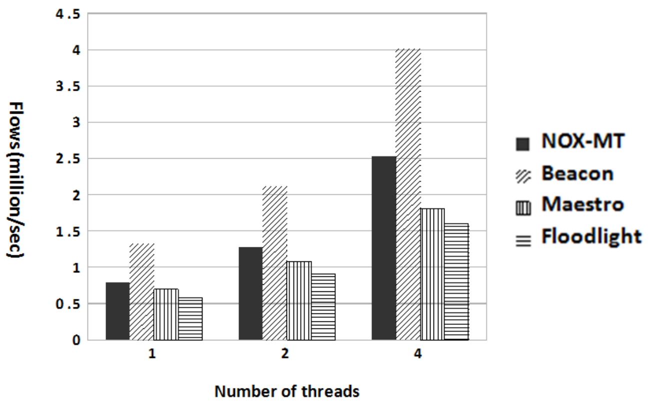


Fig. 3. Throughput performance; 32 switches, 100k MACs/switch.

message waiting to be written to the switch to reduce the per-packet latency. Static partitioning and input batching helps to improve Beacon's throughput performance in the default mode.

III. PERFORMANCE EVALUATION

In this section, we present a comparative performance analysis of the four OpenFlow controllers under consideration. We conducted experiments under many different settings and metrics. However, due to brevity constraints, we only report the most relevant results in this section, which include performance evaluation results for I/O throughput, switch scalability and latency of the controllers.

A. *Evaluation Testbed*

All of our tests were conducted on a testbed running on an HPC cluster (HP ProLiant DL160se G6 Server/ HP ProLiant DL380 G6 Server) at NUST having 34 nodes, 272 GB RAM (8 GB per node), 136 Intel Xeon processors (4 per node).[3]

We used the standard tool `Cbench` from Oflops suite [5] to evaluate the controllers. As shown in Figure 2, each `Cbench` instance in our testbed emulated a single OpenFlow switch and all of these instances sent OpenFlow `packet-in` events to a single controller. We then averaged out the total number of responses (flow-mods or packet-outs) observed on each `Cbench` instance.

In our experimental setup, `Cbench` instances ran on multiple compute nodes of the cluster, whereas the controllers ran at the head node. Each controller had 8 GB of RAM along with 4 physical cores. Compute nodes were connected to the head node via Ethernet switches with 10Gbps interconnects.

To evaluate the controllers, we ran the controllers with the standard learning switch application which performs a lookup on a table and just forwards a packet on the respective port. We chose learning switch as it is a lightweight application with a reasonable memory footprint (as long as the number of switches is not very large). Thus the learning switch application allows us to benchmark the controllers' I/O throughput rather than their computational efficiency.

To emulate a real network, we used a total of 16 compute nodes (each having 4 physical cores) for running `Cbench` instances. Each instance emulated a single OpenFlow switch sending `packet-ins` to the controller at uniform rates, hence avoiding the overhead that `Cbench` incurs when gathering the results from multiple switches within an instance. Depending on the metric being calculated, the number of `Cbench` instances running on each node was varied for different experiments as explained in subsequent sections.

B. *Throughput performance*

In this experiment, we evaluate the controllers on their ability to handle a large amount of control traffic. To achieve this, we used `Cbench` in throughput mode under which it continuously sends `packet-in` messages to the controller over a period of time and records the total number of responses received per second. We measured throughput performance of each controller with increasing number of worker threads and switches which is explained in the following subsection.

1) Thread scalability: Two `Cbench` instances were run on each node emulating a single switch, hence making a total of 32 switches all of which sent `packet-ins` to the controller with a total of $100,000$ unique source MACs per switch at a uniform rate. We measured the performance of each controller with increasing number of worker threads.

Figure 3 shows that all the controllers scale reasonably with increasing number of worker threads (t). Beacon's throughput increases from 1.35 to 4.01 million responses/sec by varying t from 1 to 4. Similarly, NOX-MT increases from 0.75 to 2.5 million responses/sec. Maestro and Floodlight provide a maximum throughput of 1.8 and 1.6 million responses/sec respectively. Beacon has the best throughput because of its static partitioning and packet batching technique. NOX-MT, on the other hand, faces a synchronization overhead introduced by the `boost:Asio`

[3]All the nodes ran Ubuntu 11.04 64-bit, kernel 2.6.38-8.

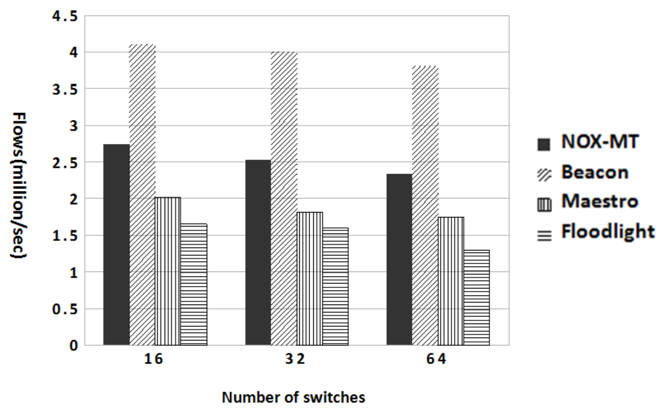


Fig. 4. Scalability with increasing number of switches; 4 worker threads.

(revealed by our profiling results) in its design, and therefore shows a reduced performance. Maestro also has a low throughput as it employs adaptive-batching technique and improves its latency by compromising its throughput performance. Floodlight being a very recent controller shows the least number of responses per second. Profiling results of Floodlight depict that the use of Java Netty for I/O relatively reduces its performance.[4]

2) Switch scalability: In this experiment, we kept a constant number of 4 worker threads for each controller, but increased the total number of switches by increasing the number of `Cbench` instances running on each node (1 instance for 16 switches, 2 for 32 switches and 4 for 64 switches) with a large volume of `packet-ins` requests/sec.

From Figure 4, it can be seen that the controllers show reasonable scalability with increasing number of switches. Again Beacon shows the best performance because of its static switch partitioning and packet batching technique and low synchronization overhead. We also see a decreasing trend in performance with increasing number of switches as the controllers are forced to handle more `packet-ins` per second due to an increase in switch density. This increase also induces a proportional increase in the size of the learning table, hence slightly increasing the table lookup time. We see that Beacon's performance is decreased from 4.10 to 3.69 million flow/sec with a degradation of 10% percent. NOX-MT, Maestro and Floodlight show a degradation of 14.9, 12.9 and 23.2 percent respectively when the number of switches is increased from 16 to 64.

C. Latency Performance

Another important feature of an OpenFlow controller is its ability to process the incoming `packet-ins` as fast as possible so that it can promptly reply to the switches. To measure controller latencies, `Cbench` was run in latency mode in which it generated a `packet-in` and waited for a response from the controller before the next `packet-in` was sent. The total number of responses/second was then counted to measure the per-packet latency of the controller. We kept a constant number of 4 worker threads and progressively increased the switch density.

[4]At the time of this evaluation experiment (March 2012), Floodlight was very recently released to the public and was undergoing significant improvements for stability and scalability.

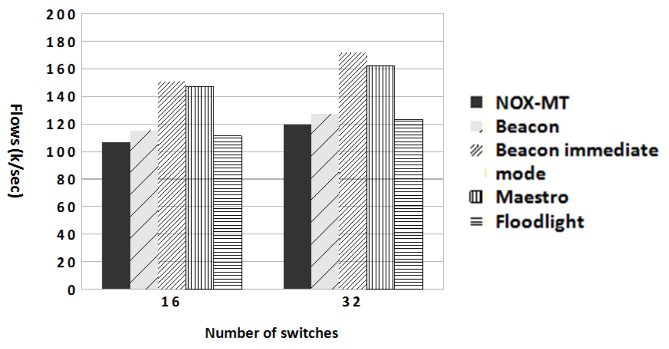


Fig. 5. Per-packet latencies of the controllers; 4 worker threads.

Maestro's performance under latency mode increases up to 162.43K responses/sec, whereas NOX-MT, Beacon and Floodlight have a maximum of 119K, 127K and 123K responses/sec, respectively. From Figure 5, we observe that Maestro has the best performance because of its workload adaptive batching technique that dynamically changes the batch sizes, thereby reducing the per-packet latency. NOX-MT, Beacon and Floodlight show a degradation in their performance as compared to Maestro because of their static packet batching design. Beacon when run in 'immediate mode' shows an improvement in the performance and the responses/sec increase up to 171.45K. Under immediate mode, Beacon attempts to perform a socket write for every outgoing OpenFlow message to improve the overall latency performance of the controller but does not maintain a balance between throughput and latency in default mode.

D. Key Findings

We summarize the key findings of our performance evaluation study in the form of the following guidelines which can be used to improve existing or design new SDN controllers:

Guideline 1: Controllers designed for high throughput (i.e., large volumes of control traffic) should use static switch partitioning and packet batching.

Guideline 2: Controllers designed for delay-sensitive control plane applications should use workload adaptive packet batching and task batching to reduce per-packet latencies.

Guideline 3: To further improve the latency performances, the controllers should send each outgoing control message individually.

IV. Combining the Architectural Guidelines in an OpenFlow Controller

In this section, we propose a multi-threaded controller which combines the performance and scalability guidelines

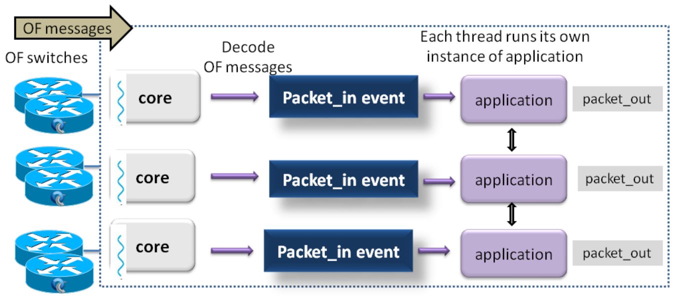


Fig. 6. Architectural Design of the proposed controller.

that we distilled in the last section. The controller focuses mainly on high performance in terms of both throughput and latency. The controller is developed in system-independent C language which removes the overhead of a higher level programming language, while also allowing the controller code to be compiled on any embedded platform seamlessly.[5]

Before presenting the performance results for the proposed controller, we discuss the key choices made in the controller's design.

A. Design Choices

1) Multi-core Support: Figure 6 shows the multi-threaded architecture of the controller. The controller executes multiple instances of an application, equal to the number of threads. Each application instance is executed in one worker thread. The number of threads to be spawned is a user configurable parameter.

Threads may maintain their own data structures according to application requirements. This can ensure their exclusive access to those data structures that can reduce or eliminate consistency and synchronization overheads. Shared data structures (if any) between multiple worker threads need to be protected by locks. Applications (either multiple instances of a single application or different applications) can communicate with each other using POSIX synchronization primitives.

2) Switch Partitioning Technique: Earlier results have revealed that static switch partitioning is the most effective technique to increase a controller's throughput. As shown in Figure 6, our proposed controller spawns n number of worker threads, where n is specified by the user. Each thread then has an equal opportunity to accept an incoming connection from an OpenFlow switch. Depending on the underlying network size, users can introduce an upper limit on the number of switch connections to an active worker thread. This method helps minimize the load imbalance and ensures that the controller provides a static granularity for assigning an equal number of switches to each worker thread.

Whenever a switch tries to establish a connection to the controller, any available thread, which has not yet reached its limit of connections, can service the switch. Once a switch is connected to a worker thread, that thread is responsible for all the subsequent I/O from that switch and the whole OpenFlow message processing chain is executed in that thread's context. This design helps to eliminate the synchronization overhead which would be incurred if multiple threads were to service a single switch. Each thread, on the other hand, can service multiple OpenFlow switches. In addition to high performance, this design choice preserves per-switch ordered packet processing in the control plane.

3) Packet Batching Technique: Static packet batching is used, in which a worker thread reads a fixed number of bytes from a socket each time the data is available to prevent frequent system calls. The requests obtained from each socket form a batch and are then processed, thereby improving the overall throughput performance of the controller.

Packet Processing: Once the packets are received in the form of batches, per packet processing is done, i.e. each packet is processed individually. In the processing stage, every OpenFlow message is decoded individually. The controller makes a system call for every outgoing message individually, in order to reduce the latency and keeps a balance between throughput and latency performances.

B. Comparative Performance Evaluation

We now present the performance results of the proposed controller in comparison to other OpenFlow controllers. We used exactly the same evaluation setup that was described earlier.

TABLE I
COMPARISON BETWEEN THREAD SCALABILITY (IN MILLION FLOWS/SEC)

	Threads		
	1	2	4
Proposed OF controller	1.36	2.62	4.50
Beacon	1.35	2.12	4.01

Table I shows that the proposed controller's throughput performance at 4 threads is 4.50 million flows per second which is about 11.5% greater than the highest performing controller - Beacon - having 4.0 million flows/sec. The almost linear scaling in the performance of the proposed controller (1.36 - 4.50 million flows/sec) is a consequence of an efficient multi-threaded architecture, minimal synchronization overhead, static switch partitioning technique along with reading a fixed batch of packets from the underlying switch.

As shown in Table II, while moving from 16 to 64 switches, the proposed controller shows a degradation of 7.3% whereas Beacon degrades to about 10%. The controller shows its ability to scale with the increasing size of the underlying network showing an improvement of 2.7% as the switch density is increased. Again the efficient

[5]To allow the community to benefit from the results of this study result, we will be releasing the controller code in open-source after paper acceptance.

TABLE II
Comparison between switch scalability (in million flows/sec)

	Switches		
	16	32	64
Proposed OF controller	4.52	4.50	4.19
Beacon	4.10	4.01	3.69

I/O loop architecture of the controller and its minimal memory overhead contribute to its stable performance.

Latency measurements shown in Table III indicate that the proposed controller performs significantly well in terms of efficient packet processing. Under latency mode, the performance of the controller increases upto 197K flows/sec with an improvement of 11.6% in comparison to the highest performance among existing controller. This high output can be mainly attributed to efficient per-packet processing design and the controller's ability to write an output message for each packet individually.

TABLE III
Comparison between latency performance at 4 threads (in k flows/sec)

	Switches	
	16	32
Proposed OF controller	173.13	197.50
Beacon immediate mode	150.67	171.45
Maestro	147.35	162.43

V. Conclusion

We presented a detailed architectural evaluation of four publically-available OpenFlow controllers, focusing on strengths and weaknesses of their architectural design choices. Our results revealed promising guidelines that can be used to attain consistent improvements in the performances of SDN controllers. We also presented a controller based on the derived guidelines. The proposed controller is shown to have consistent improvements in terms of throughput, latency and switch scalability.

References

[1] "Nox homepage," http://noxrepo.org/.
[2] "Beacon homepage," https://OpenFlow.stanford.edu/display/Beacon/Home.
[3] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable openflow contro," Rice University, Tech. Rep., 2010.
[4] "Floodlight homepage," http://floodlight.OpenFlowhub.org/.
[5] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "Oflops: An open framework for openflow switch evaluation." in *PAM*, ser. Lecture Notes in Computer Science, N. Taft and F. Ricciato, Eds., vol. 7192. Springer, 2012, pp. 85–95. [Online]. Available: http://dblp.uni-trier.de/db/conf/pam/pam2012.html#RotsosSUSM12
[6] Y. G. M. C. R. S. Amin Tootoonchian, Sergey Gorbunov, "On controller performance in software-defined networks," in *HOT-ICE*, 2012.
[7] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: Balancing fairness, latency and throughput in the openflow control plane," Rice University, Tech. Rep., December 2011.
[8] "Boost::asio documentation," http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio.html.