Maestro: Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane

Zheng Cai   Alan L. Cox   T. S. Eugene Ng
Department of Computer Science, Rice University

Abstract

The fundamental feature of an OpenFlow network is that the controller is responsible for the configuration of switches for every traffic flow. This feature brings programmability and flexibility, but also puts the controller in a critical role in the performance of an OpenFlow network. To fairly service requests from different switches, to achieve low request-handling latency, and to scale effectively on multi-core processors are fundamental controller design requirements. With these requirements in mind, we explore multiple workload distribution designs within our system called Maestro. These designs are evaluated against the requirements, together with the static partitioning and static batching design found in other available multi-threaded controllers, NOX and Beacon. We find that a Maestro design based on the abstraction that each individual thread services switches in a round-robin manner can achieve excellent throughput scalability (second only to another Maestro design) while maintaining far superior and near optimal max-min fairness. At the same time, low latency even at high throughput is achieved thanks to Maestro's workload-adaptive request batching.

1 Introduction

The emerging OpenFlow [16] network architecture allows rich networking functions to be directly programmed on a controller platform, which in turn sends instructions to switches to carry out the functions. OpenFlow switches have not only been widely used by the research community, such as the GENI project [2], but have also attracted significant commercial interest. A large number of computing and networking companies have joined the Open Networking Foundation [4] to help standardize and commercialize OpenFlow. The success of OpenFlow can also be seen from the large number of recent use cases: programmable network testbeds [14][25][20], datacenter network designs [22][19][5], enterprise network designs [18][17][11], network measurement systems [6][23], to name just a few recent examples.

1.1 The performance challenge

A fundamental feature of OpenFlow is that the controller is responsible for making control decisions for every traffic flow in the network. Whenever a switch sees a flow's first packet, because there is no flow entry configured on the switch's flow table to match this flow, the first packet will be forwarded to the controller. We call this first packet a "flow request". The controller runs user defined applications to process a flow request, for example the controller computes a path for this flow and installs flow entries on every switch along the chosen path, so that subsequent packets of this flow can be handled by the switches locally. Finally, the flow request packet itself is sent back to the origin switch.

Optimizing the performance of the controller system is critical if OpenFlow were to be successful in high-end deployment scenarios such as warehouse-scale datacenters and large enterprises. Recent measurements of traffic in datacenters of various sizes and purposes [7] have shown that, in datacenter deployments, the controller could see up to 0.1 million flow requests per second per server rack today.

To address the performance challenge requires a multi-prong approach: (1) maximize the performance of each physical controller machine; (2) enable a cluster of controller machines to work as a single logical controller; (3) partition the network into zones with separate controllers. While all three directions are equally important and are being investigated, this study focuses on the first direction. In particular, *we investigate what software design strategies would optimize the performance of a controller machine under the workload characteristics of OpenFlow, assuming the hardware is a commodity computer based on a modern multi-core processor architecture.*

1.2 Fundamental requirements

Optimizing the performance of a controller means more than just hitting the highest aggregate flow request handling throughput. A controller that does so but unintentionally starves some subset of requests is useless. More generally, a controller that has arbitrary performance bias against certain requests is undesirable. A controller that achieves high throughput but has uncontrollable latency is also undesirable. Optimizing performance requires a balance between fairness, latency, and throughput.

Fair capacity allocation: The capacity of the controller must be "fairly" allocated among source switches that generate requests according to a well defined fairness policy. Especially when the offered workload is larger than the capacity of the controller, the controller must not arbitrarily favor certain sources. A reasonable fairness policy is weighted max-min fairness, where the weights are specifiable by the operator. Equal weights can be assigned to realize a basic max-min fairness policy.

Controllable latency: A controller's throughput in general can be improved by sacrificing latency. For instance, the overhead of a socket read system call can be amortized across a larger number of pending requests by using a larger read buffer, thereby increasing throughput. Many strategies of this nature could generally be called batching. Batching, however, increases the latency experienced by requests that are positioned early in the batch. Furthermore, batching could also hurt fairness at the fine timescale, resulting in higher request handling latency even for a switch that originates requests at a low rate. An optimized controller must restrain latency while pursuing high throughput.

Scalable throughput on multi-core: The controller must be able to run multiple copies of user applications in parallel to scale up throughput on multi-core processors, and must do so while maintaining fairness and controllable latency. Users of the system must have the option to write simple single-threaded applications and leave it to the controller to parallelize them. This option reduces the complexity of the application programs that users have to write, thereby improves user productivity and system robustness.

1.3 Contributions

We present an open source controller platform named Maestro [3]. Maestro provides the low level interfaces for interacting with an OpenFlow network, enables the composition and concurrent execution of user applications, and ensures the consistent usage and update of shared data. This study uses the Maestro platform to investigate controller software design choices.

We present four workload distribution designs termed shared-queue, static-partition, dynamic-partition, and round-robin. These designs represent different trade-off points between complexity, fairness, and scalability. We also compare Maestro designs against two other available controllers, NOX [13] and Beacon [1], both currently employ a static-partition design. Through extensive experimental evaluation, we find that the round-robin design achieves far superior and near optimal fairness while having excellent scalability, second only to the dynamic-partition design.

We present a workload-adaptive request batching algorithm that automatically selects the granularity for batching requests for improved throughput while ensuring request handling latency is well controlled. The key to the algorithm is to use actual throughput and latency measurements at run-time to control the dynamic adaptation. Experimental results show that the algorithm is very effective at maintaining high throughput while restraining latency regardless of the workload. In contrast, the static batching algorithm currently employed by NOX and Beacon leads to unnecessarily large latency at heavy load.

Together, our designs, algorithms and experimental evaluations provide extensive and quantitative insights on balancing fairness, latency, and throughput in the OpenFlow control plane.

1.4 Paper organization

The rest of this paper is organized as follows. We discuss the related work in Section 2. In Section 3, we investigate multiple design choices in Maestro to address the fundamental requirements. In Section 4, we experimentally evaluate the performance of different Maestro designs, together with other available OpenFlow controllers. We conclude in Section 5.

2 Related work

NOX [13] and Beacon [1] are both multi-threaded, programmable OpenFlow controllers developed in parallel to Maestro. NOX, Beacon and Maestro all allow users to write simple single threaded applications and can run them in parallel to scale up throughput on multi-core processors. While there are far too many design and implementation differences between NOX, Beacon and Maestro to enumerate, a focused comparison with respect to the way they distribute the request workload among worker threads could be made. In this regard, NOX and Beacon turn out to be quite similar. NOX and Beacon both statically assign the requests from a fixed subset of the network switches to each worker thread. This design maximizes parallelism and is conceptually ideal when requests are uniformly arriving from all switches. However, as we experimentally show, because not all worker threads run at exactly the same rate in practice, even under a uniform workload, there could be arbitrary performance bias. And when the workload is not uniform, this design suffers from poor fairness and potentially suboptimal throughput due to the under-utilization of some worker threads. NOX and Beacon both adopt a static granularity for request batching for improving the throughput of an individual worker thread, though the actual batch sizes used do differ. Although both systems achieve impressive raw aggregate throughput, as expected, such a static batching strategy leads to unnecessarily large request handling latency when the system is under heavy load. We hope the solutions that we present within Maestro for balancing fairness, latency and throughput could inform future development of NOX and Beacon.

The design of our solutions in Maestro has been somewhat influenced by recent works in multi-core software router design [10, 9]. Whenever appropriate, Maestro liberally borrows from the insights from these works, such as the importance of batching workload, and the importance of minimizing cross-CPU-core synchronization overhead and cache contention overhead.

Complementary to solutions that aim at maximizing the performance of each physical controller machine, several recent works have aimed at enabling a cluster of controller machines to work as a single logical controller to further improve scalability. HyperFlow [24] extends NOX into a distributed control plane. By synchronizing network-wide state among distributed controller machines in the background through a distributed file system, HyperFlow ensures that the processing of a particular flow request is localizable to an individual controller machine. Onix [15] further provides a general framework for building distributed coordinating network control plane, especially for the case of OpenFlow controllers. More specifically, Onix provides a Network Information Base which gives users access to several state synchronization frameworks with different consistency and availability requirements.

DIFANE [26] presents another approach to improve flow-based networks' control plane performance. It provides a way to achieve efficient rule-based policy enforcement in a network by performing policy rules matching at the switches themselves. DIFANE's network controller installs policy rules in switches and does not need to be involved in matching packets against these rules as in OpenFlow. However, OpenFlow is more flexible since its control logic can realize behaviors that cannot be easily achieved by a set of relatively static policy rules installed in switches. Ultimately, the techniques proposed by DIFANE to offload policy rules matching onto switches and our techniques to increase the performance of the controller are highly complementary.

3 Maestro: Addressing the fundamental requirements

In this section, we explore multiple design choices for addressing the fundamental requirements.

3.1 Overview of the Maestro system

Maestro provides the low level interfaces for interacting with an OpenFlow network, such as the "chopping", "parsing", and "output" stages shown in Figure 1 & 2. Because the length of each OpenFlow packet is specified

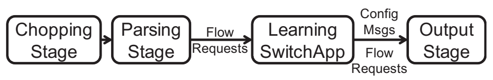


Figure 1: Learning switch functionality

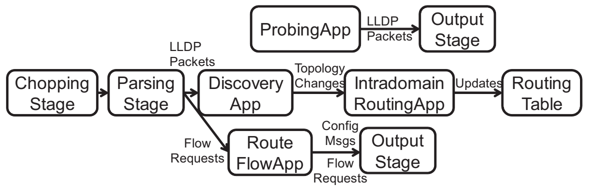


Figure 2: Routing functionality

in its header, the "chopping" stage is responsible for correctly chopping raw bytes read from a stream socket into correctly aligned individual OpenFlow packets. Since a socket read operation could receive an incomplete OpenFlow packet, the "chopping" stage for one socket cannot be parallelized, and lock synchronization must be used for socket read to ensure the correctness of "chopping". On the other hand, the "parsing" stage which parses raw OpenFlow packets into specific messages such as flow requests, can be parallelized. The "output" stage puts outgoing data into OpenFlow format, and sends out to destination switches. If multiple threads are writing to the same socket, synchronization is also needed.

Users of Maestro write their own applications, and use the provided user interface to configure their execution sequences to realize different functionalities. Figure 1 shows the "Learning Switch" example. There is only one application `LearningSwitchApp`. This application first remembers the switch port from which a request came from and associates the source address of the request packet to that port. It then checks to see if the destination address of the request packet has been associated to a port before. If so, it installs a flow table entry at the origin switch for forwarding that destination address to that port; subsequent packets for that destination can be directly handled by that switch. Otherwise, the controller instructs that switch to flood the request packet along a spanning tree maintained by the switch.

Figure 2 shows the "Routing" example. In the first user-defined application sequence, `ProbingApp` periodically sends out LLDP packets to all active ports of each connected OpenFlow switch. As shown in the second application sequence, these LLDP packets will be sent back to Maestro by the neighbor switches connected to these ports, and `DiscoveryApp` processes these packets to know the topology of the network. Based on such topology information, `IntradomainRoutingApp` calculates the `RoutingTable`, which is used by `RouteFlowApp` in the third application sequence, to calculate the entire path for incoming flow requests.

Maestro also provides a user interface for specifying applications, such as `LearningSwitchApp` and `RouteFlowApp`, to be parallelized by Maestro, so that users only need to write single-threaded application but can still achieve high performance. Depending on the number of available CPU cores in the system, Maestro dynamically creates multiple worker threads, to work on multiple instances of the parallelized application. Each application instance is executed in one worker thread to process a portion of the incoming flow requests. In addition, Maestro adopts standard techniques to ensure the consistency of shared state among concurrent applications. For example, when `IntradomainRoutingApp` updates the `RoutingTable` at run-time, Maestro stalls pending `RouteFlowApp` instances until the `RoutingTable` updates finish. Unfortunately, due to space constraint, we refer the readers to a technical report [8] for more details about Maestro that have to be left out here. Note that the source code for Maestro is available for download [3].

3.2 Achieving fair capacity allocation while having scalable throughput

The offered workload needs to be distributed among all available CPU cores in order to maximize the system's throughput. How such distribution is done will directly affect the throughput scalability, and at the same time the fairness in allocating the capacity of the system.

3.2.1 Maestro-Shared-Queue

To achieve a basic max-min fair allocation of the capacity of the system to all source switches, the controller needs to give each switch an equal chance to be served. Initially in [8] we started with a straight-forward design, in which Maestro has a dedicated thread which is responsible for reading incoming bytes from socket buffers. This thread uses a mechanism which is similar to "select()" in the Berkeley sockets API to select all sockets that have pending bytes, performs socket read on all of them with the same maximum read size, and chops the raw bytes into raw OpenFlow packets. We call this thread a "select thread". All the raw OpenFlow packets are put into a queue shared by all the worker threads. We call this design Maestro-Shared-Queue from now on. The worker threads fetch raw OpenFlow packets from the shared queue, parse them into OpenFlow messages, and execute applications to process them. workload is evenly distributed among all worker threads, because any idle worker thread will always be able to pick up pending raw OpenFlow packets from the queue if there is any available.

This design theoretically can achieve a max-min fair allocation of the system's capacity, because the select thread is giving each source switch equal chance (in terms of bytes) to be served. If all the flow requests have the same number of bytes, which is the case for TCP syn packets, each switch will also get equal service in terms of the number of flow requests served. More generally, to achieve weighted max-min fairness, a source with weight w will be given w chances to be served in each round. Although simple, this design has fundamental drawbacks, especially in throughput scalability. First of all, all worker threads have to share a request queue, so they have to rely on lock synchronization which introduces a non-trivial amount of overhead. Second, reading and chopping of raw bytes for a flow request is done by a different thread from the worker thread that handles the remaining parts of the processing, which can lead to extra cross-CPU-core overhead. Third, one single select thread can only process a certain amount of requests per second. If the worker threads' aggregate processing capacity exceeds this dedicated select thread's, either the throughput of the system becomes bottle-necked, or additional select threads need to be added. The next design choices avoid having dedicated select threads.

3.2.2 Maestro-Static-Partition

To eliminate the overhead introduced by lock synchronization of concurrent read accesses to a switch socket, switch sockets can be partitioned and assigned to specific worker threads, so that each worker thread has exclusive read access to switch sockets in its partition. This also minimizes the cross-CPU-core overhead because each flow request is processed entirely by a worker thread (assuming that each worker thread is bound to a specific CPU core, which we will discuss in more details in Section 3.2.5). This is the design chosen by NOX and Beacon. We also explore this design in Maestro and name it Maestro-Static-Partition. Usually each worker thread is assigned the same number of switch sockets to balance the workload among all worker threads. However, because each switch can have a different flow request arrival rate (which we call the "input rate" from now on), an equal number of assigned switches does not mean equal workload assignment. As a result, such static partitioning may not be able to evenly distribute the workload among all worker threads, so both the fairness and throughput of the system will be affected.

```
For each worker thread $t$
  Set $t.assigned = 0$
  Put $t$ into $minHeap$ sorted by $t.assigned$
Sort all switches $sw$ based on $sw.inputRate$, from high to low
For each $sw$ in sorted list
  Assign $sw$ to worker thread $t$ at $minHeap.top()$
  $t.assigned$ += $sw.inputRate$
  $update(minHeap)$
```

Figure 3: Re-partitioning algorithm

3.2.3 Maestro-Dynamic-Partition

To improve upon Maestro-Static-Partition, we can dynamically divide switches into **n** partitions, where **n** is also the number of worker threads. To fully utilize all worker threads in the system, the dynamic partitioning needs to be done effectively so that the workload is evenly distributed among all worker threads. First of all, we need to measure the recent input rates of the switches, in order to predict the future input rates for dynamic re-partitioning. The caveat is that this assumes the input rates are stable over a short timescale. Such measurement and re-partitioning can neither be done too frequently because each re-partitioning involves unavoidable lock synchronization overhead, nor can they be done too infrequently because the measurement based prediction and re-partitioning could be much less accurate. Second, the re-partitioning itself is a NP-complete problem to solve optimally [12]. In this study, we adopt a simple greedy algorithm as shown in Figure 3.

We call this design Maestro-Dynamic-Partition. Even if input rates can be reasonably predicted, this design still has other limitations. First, max-min fairness in system capacity allocation in general cannot be achieved even if each worker thread makes sure that all switches within its partition receive equal chance of being handled. For example, suppose there are 2 worker threads and 3 switches with input rates r, r, and $2r$ respectively. Switch 1 and 2 are therefore assigned to thread 1 while switch 3 is assigned to thread 2. In this case, switch 1 and 2 can receive only up to 25% of the system capacity, while switch 3 can receive up to 50%. Second, if the workload cannot be evenly partitioned among worker threads, CPU cores may not be fully utilized, thus throughput will not be maximized. We will show in Section 4 that the fairness problem and the CPU core under utilization problem, despite being less severe than that in Maestro-Static-Partition, still exist.

3.2.4 Maestro-Round-Robin

A fourth design choice we consider is called Maestro-Round-Robin. In this design, each worker thread is individually running a round-robin service loop among all switch sockets. By doing this, each switch will be given equal chance to be serviced by each worker thread. Thus, conceptually, the overall system also gives equal chance to each switch and achieves max-min fairness, or weighted max-min fairness by giving a switch w chances to be served per round per thread. However, due to the limitation that only one worker thread can read bytes and perform chopping for a switch at a time, each worker thread needs to check whether another thread is already performing reading and chopping on a switch socket. This leads to some locking overhead which affects the throughput of the system. We will show the trade-off between fairness and throughput achieved by Maestro-Round-Robin in Section 4.

In Maestro-Round-Robin, each flow request is processed entirely by one of the worker threads, thus cross-CPU-core overhead is also minimized. Because each worker thread can process requests from all switches, Maestro-Round-Robin can have better throughput than Maestro-Dynamic-Partition in the cases where the workload cannot be evenly partitioned. Furthermore, when one worker thread finds out that another thread is performing chopping on a switch, the worker thread skips this switch and tries the next switch, to prevent wasting CPU cycles waiting for another thread to finish. However, such skipped switches need to be remembered, so that before a worker thread finishes one round, these skipped switches are revisited, so as to give each switch an equal chance to be serviced.

3.2.5 More on request and thread bindings

As alluded to earlier, minimizing cross-CPU-core overhead is critical to maximizing the throughput. Some experimental results can be found in our previous work [8], so here we only describe our findings briefly. First of all, binding threads to cores is necessary, because otherwise there will be a huge overhead introduced by thread context switch if the operating system moves the execution of one worker thread to another CPU core at run-time. Second, it is also important to bind requests to threads, so that each flow request is processed as much as possible by the same worker thread. Such binding minimizes the overhead introduced by data synchronization between threads. Recent work in multi-core software router design has shown that in some cases, it is better to have each thread working for one small processing step because this could reduce the cache misses of a thread [9]. We leave it as future work to explore whether this design model could be borrowed in Maestro.

3.3 Achieving controllable latency while having high throughput

There is unavoidable overhead in system calls such as socket read/write, in executing applications to process flow requests such as preparing the state environment for applications, warming up the CPU caches, etc. As a result, amortizing such unavoidable overhead across multiple requests is critical for improving the throughput of the system. Such overhead amortization can be done by reducing the number of system calls by reading/writing more bytes per each socket call, and reducing the number of application executions by having an application process a batch of requests in one execution.

Both NOX and Beacon adopt this approach: each worker thread tries to read up to a large number of bytes (we call this the "maximum read size") from a socket each time. The requests obtained from *each socket read* forms a batch. Note that the size of each batch therefore depends on the amount of data pending at a socket at the time of the read. The thread then processes all the requests in the batch, and writes all pending messages for a switch by calling socket write once when the destination socket is write-ready. The maximum read size is static and the result depends a lot on the value chosen. To provide a comparison, we also configure Maestro-Static-Partition to perform a large socket read, and let the application processes all requests generated from a socket read as a batch.

In the other three Maestro designs, we adopt a different approach for amortizing the overhead that provides much more control over the batching behavior. First of all, we use a much smaller maximum read size in socket reads than NOX and Beacon. Although this means more system call overhead, it provides much finer grained control over system latency because the system can visit and serve each switch more frequently. Second, a worker thread batches up to a certain number of flow requests, as determined by an automatically selected parameter called the Input Batching Threshold (IBT), before it initiates applications to process all flow requests in the batch at once. Thus, the size of a batch is *independent* of the amount of pending data at individual sockets. Furthermore, the requests in a batch could very well come from multiple socket reads from different switch sockets. Finally, similar to NOX and Beacon, all the messages to the same destination generated from processing a batch are also sent to the destination by calling socket write only once when the socket is write-ready.

The key question then is, how should the IBT value be chosen? When the IBT is increased, on one hand, throughput could theoretically become higher because the overhead is further amortized. On the other hand, in reality the throughput does not keep growing with ever larger IBT, because as more memory is used to form the batch, memory access efficiency decreases and at some point it will out-weight the overhead amortization gain. In addition, with a larger IBT, flow requests will experience longer latency in the system. However, if the IBT is too small, not only the throughput of the system will be low, but also the latency will increase because the low throughput increases the waiting time of the flow requests in socket buffers. Furthermore, for different aggregate input rates, the system needs different IBT values to achieve a good balance between high throughput and low latency. Thus, what we need is an IBT adaptation algorithm according to the dynamic input rate of the workload.

```
**Initialization:**
  $Trend = increasing$
  $IBT = 10$ (always lower bound by 10)
  $S_n$ initialized directly to $S$ in first use
  $S' = 0$ in first use

**After finishing one full IBT-sized batch:**
  Let $t$ = time spent in processing this batch
  Let $n$ = size of this batch, score $S = n/t$
  Smoothed score $S_n = (1-w) * S_n + w * S$
  Let $S'$ be the smoothed score of last full IBT-sized batch
  If ($S_n \leq S'$)
    $Trend = reverse(Trend)$
  If ($t > BatchingDelayUpperbound$)
    $Trend = decreasing$
  If ($Trend == increasing$)
    $IBT$ += 10
  Else
    $IBT$ -= 10

**When no pending bytes left in any socket buffer:**
  Process the current batch ignoring IBT
  $Trend = decreasing$
  $IBT$ -= 10
```

Figure 4: IBT adaptation algorithm

Each worker thread independently uses the IBT adaptation algorithm in Figure 4 to maximize throughput while restraining latency. The algorithm measures the time spent in the processing of a full IBT-sized batch, and calculates the throughput score S of this batch. To eliminate noise from the measurements, the algorithm maintains a smoothed average score $S_n = (1-w)*S_n+w*S$, where S_n is the smoothed score for batch size n. Currently we use a weight of $w = 0.2$. The algorithm compares the smoothed throughput score of this batch to that of the last full IBT-sized batch. If the score is higher, the algorithm keeps the current IBT adjustment trend; otherwise, the trend is reversed. The IBT is adjusted by a fixed amount each time, currently chosen to be 10 requests.

The algorithm uses the $BatchingDelayUpperbound$ (BDU) parameter to control the latency of the system. When the IBT adaptation algorithm finds the time spent in one batch exceeds the BDU, the trend is directly set to decreasing. The BDU can be dynamically configured by the user of Maestro. So if she can tolerate a higher latency, Maestro will operate at higher IBT to achieve a higher throughput. If she requires a tighter in-system latency, she can set a low BDU, at the cost of potentially lower throughput. Notice that although related, BDU cannot be directly translated into end-to-end latency. Maestro cannot control the latency outside of itself, such as the round-trip network propagation delay, the delay in socket buffers, or the delay introduced by the kernel. In addition, BDU only controls the latency of one batch, so if there are a large number of switches to be served, a flow request from one switch may have to wait for more than one batch.

Finally, under light load, when the algorithm finds there is no pending bytes in any of the sockets, the algorithm releases the current batch for immediate processing ignoring the current IBT, decreases the IBT, and sets the trend to decreasing. The effectiveness of the IBT adaptation algorithm is evaluated in Section 4.3.

4 Evaluation

In this section, we evaluate the current Maestro prototype (version 0.2.1, implemented in Java, available at [3]) under a number of different workload scenarios to see how effectively our design choices address the fundamental requirements. Also, we compare Maestro against a multi-threaded version of NOX written in C++ (branch destiny-fast, lead by Amin Tootoonchian), and Beacon [1], which is also written in Java. Through these comparisons, we aim to quantify the overall strengths and weaknesses of the Maestro designs, as well as the costs and benefits of our specific design choices. Because we are more interested in the raw performance of the system itself than the applications, in all of the experiments, we run only the most simple "Learning Switch" application.

Both NOX and Beacon statically assign switches to worker threads, and use a static maximum read size for batching. Specifically, NOX uses 512KB for its maximum read size, while Beacon uses 64KB. In order to provide a close approximation to Beacon's static approach, in Maestro-Static-Partition we also use 64KB as the static maximum read size. While in all other Maestro designs we use 2KB as the static maximum read size. In this section, we will show how these static design choices affect the fairness, latency, and throughput of these systems, as compared to the dynamic approach of Maestro.

4.1 Experiment setup and methodology

Instead of using the standard controller benchmark "cbench" provided by the OpenFlow community, we have implemented and use our own network emulator. Our network emulator provides greater functionality than cbench. It can not only emulate the functionality of the OpenFlow switch's control plane, but also generate flow requests at different controlled rates for the emulated switches. This additional feature enables us not only to precisely measure how fairly the capacity of the controller is allocated among all switches, but also to evaluate the performance of the controller under different workload scenarios.

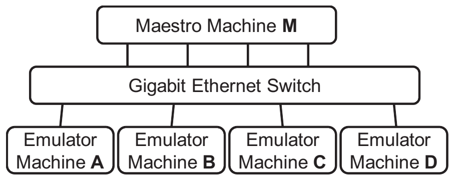


Figure 5: Experiment platform setup

In each experiment, the OpenFlow controller is running on a server machine with two Quad-Core AMD Opteron 2393 processors (8 cores in total) with 16GB of memory. Because there are other processes/threads responsible for managing either the Java virtual machine (such as class management and garbage collection), or serving other system functionalities, we dedicate at least one processor core for such work, while the remaining 7 cores are used by the controller for worker threads. Thus the best throughput (for most of the cases) is achieved with 7 worker threads on this 8 core server machine. This machine has four 1Gbps NICs to provide enough network bandwidth. The controller machine is running Ubuntu 9.10 with a 2.6.31 Linux kernel and the 64-bit version of JDK 1.6.0_25.

We run the emulator simultaneously on four machines to provide enough CPU cycles and network bandwidth for the emulation, as shown in Figure 5. Each of the emulator machines is connected to a gigabit Ethernet switch by a 1Gbps link. Each of these machines emulates one fourth of all the OpenFlow switches in the emulated network. We run experiments using both a 79-switch and 1347-switch topology [21], to evaluate the effect of network size. Together, the four machines can generate up to four million flow requests per second. Additionally, the emulator allows us to control the distribution of these requests in terms of which switch they originate from.

We use three primary metrics for measuring the performance of the controllers. The first one is the throughput of the controllers, measured in requests per second (rps), for which a larger value is better. The second one is the average delay experienced by a low-rate (5rps) probing switch, measured in milliseconds, for which a smaller value is better. This delay is the end-to-end delay measured by the emulator. We choose not to use the average delay experienced by all requests, because the delay of requests from heavy-rate switches is largely affected by the underlying TCP socket read/write buffer size configuration, which could vary significantly across different

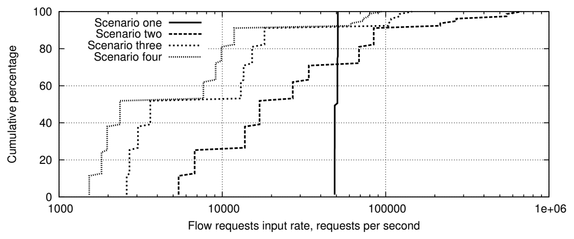


Figure 6: Distribution of flow request rates

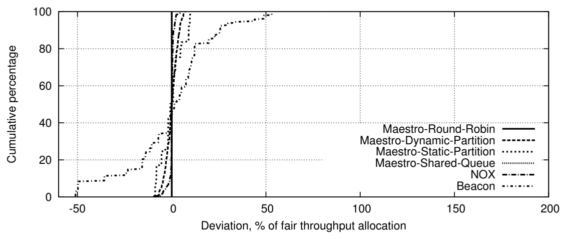


Figure 7: Fairness result of scenario one

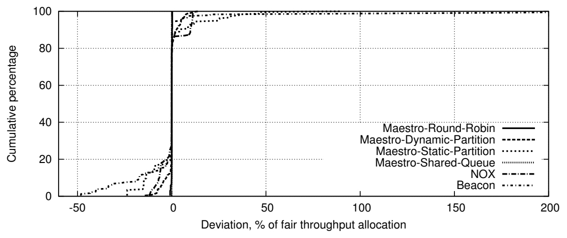


Figure 8: Fairness result of scenario two

systems. Instead the average delay of a low-rate probing switch is a more accurate measurement of the latency introduced by the controller plus the round trip time, because the TCP socket read/write buffer of the probing switch will be empty most of the time. The third one is the fairness of the capacity allocation. To measure the fairness, we first calculate the max-min fair share of the capacity for each switch, given each switch's request rate and the controller's total throughput. Then we calculate the deviation of the actual share that each switch receives from its fair share. Finally we plot the CDF of such deviations. A deviation distribution around 0 means very good fairness, while a wider deviation distribution means worse fairness.

4.2 Fairness of capacity allocation

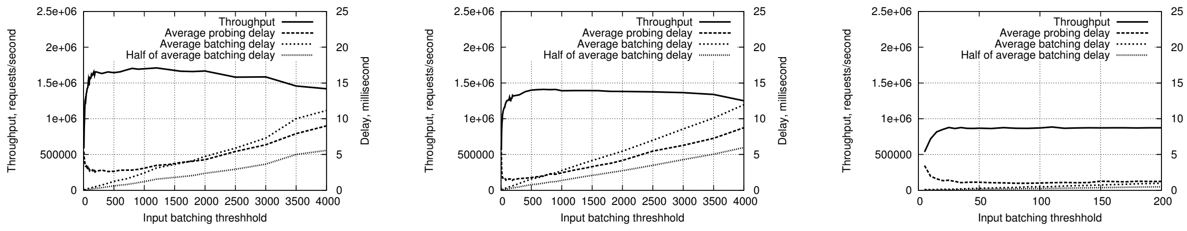
In this section, we compare the fairness of capacity allocation for all Maestro designs (Maestro-Round-Robin, Maestro-Dynamic-Partition, Maestro-Shared-Queue and Maestro-Static-Partition) against NOX and Beacon, through two different scenarios. We use the 79-switch topology instead of the 1347-switch one, because there is less fluctuation when the emulators are generating requests for fewer switches, so that the fairness measurement is more accurate. In all of these experiments, we run the controllers with four worker threads, not only to ensure that the server machine with eight cores can provide enough CPU cycles for the controller, but also to make sure that the capacity of the controller is always below the aggregate request rate from the emulators at any instant in time. Otherwise, 100% of the requests could be handled which leads to a naturally fair allocation.

In the first scenario, each emulator tries to generate flow requests for its emulated switches at uniform rates. However, because the four emulators cannot be perfectly synchronized while at the same time providing a high request rate, the switches from different emulators do not have exactly equal request rates. The distribution of request rates is shown as scenario one in Figure 6. An optimal fair capacity allocation will be that all switches get about the same share of the system's throughput. As shown in Figure 7, both Maestro-Round-Robin and Maestro-Shared-Queue achieve very good fairness in capacity allocation. All of the other designs that assign switches to worker threads have worse fairness, especially Beacon which can allocate up to 50% less or more throughput to some switches from their fair share. This is because not all worker threads can process requests at exactly the same rate, even in this simple scenario where work load can be evenly distributed among worker threads, there is still arbitrary fairness bias introduced.

Next, we configure the emulators to generate requests for switches with vastly skewed request rates shown as scenario two in Figure 6. This is a more challenging scenario for all of the controllers. As shown in Figure 8, Maestro-Round-Robin and Maestro-Shared-Queue again have the best fairness performance, with all deviations smaller than 1%. On the other hand, all other controllers have worse fairness. We can see that the deviations are much worse at the tails because the switches which generate heavier rates of requests get much larger shares than is fair. Again Beacon has the worst fairness, where up to 200% more throughput is allocated to some source switches than is their fair share.

4.3 Effectiveness of the IBT adaptation algorithm

In this section, to evaluate the effectiveness of the IBT adaptation algorithm, we focus on Maestro-Round-Robin using four worker threads and running on the 79-switch emulated network with skewed request rates. To establish the baselines and to investigate the effect of different IBT values on the throughput and delay of the sys-



(a) 4 million rps request rate (b) 1.4 million rps request rate (c) 0.85 million rps request rate

Figure 9: Results of using different static IBT values

tem, we manually measure the performance of Maestro using different static IBT values under different workloads. We choose three different workloads: 4 million rps as in Figure 6's scenario two, which is more than twice the maximum throughput of Maestro at four worker threads; 1.4 million rps as in Figure 6's scenario three, which is about 80% of the maximum attainable throughput; and 0.85 million rps as in Figure 6's scenario four, which is about 50% of the maximum attainable throughput.

As shown in Figure 9, under the 4 million rps workload, when we keep increasing the IBT, the throughput of Maestro grows at first, but starts to decrease when the IBT is larger than 1200. The probing delay decreases at the very beginning because of the significant improvement in throughput. Then the probing delay keeps growing with larger IBT, and is about half of the batching delay plus the extra round trip time outside Maestro. This is because in this 79-switch network where there are not too many sockets to read from in a single round, and each time we read at most 2KB from a socket, the batching delay is essentially the worse case delay for a request to spend within a batch, so the average case is that a request spends half of the worse case delay in the batch. A 3ms BDU would translate to a maximum IBT of about 1100. For the 1.4 million rps workload, when the IBT is larger than 500, the throughput starts to flatten out and decrease slowly. For the 0.85 million rps workload, an IBT value of 25 is sufficient for Maestro to handle every request of the offered 0.85 million rps, while keeping the probing delay very low. In this case of light load, the BDU should not be reached by the algorithm.

Now, we enable the IBT adaptation algorithm in Maestro-Round-Robin, set the BDU to 3ms, and conduct an experiment where the aggregate request rate dynamically changes over time. In this experiment, the aggregate request rate offered by the emulator changes every ten seconds. It starts at 4 million rps, then drops to 1.4 million rps, then drops again to 0.85 million rps, then goes back to 1.4 million rps, and finally returns to 4 million rps. Through this dynamic configuration we want to show that the IBT adaptation algorithm can effectively

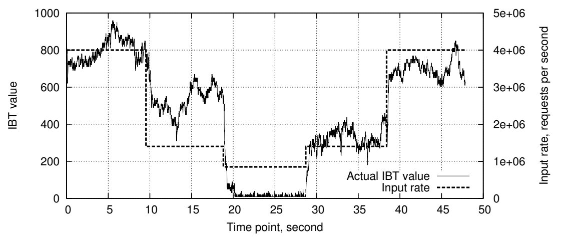


Figure 10: Dynamic IBT under changing request rate

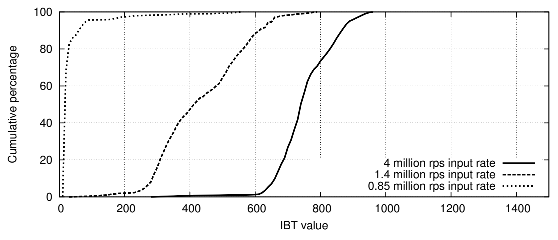


Figure 11: IBT distribution upon different request rate

handle both an increasing and decreasing aggregate request rate. Figure 10 shows the dynamic IBT values generated by the adaptation algorithm over time, together with the corresponding aggregate request rate. In this figure we can see that, first, although IBT values generated by the adaptation algorithm is fluctuating, Maestro is operating at reasonable IBT values (within peak throughput area) in all regions, while at the same time keeping not only the batching delay but also the end-to-end probing delay under 3ms (as shown in Table 1). Second, the adaptation algorithm responds to changes in the workload reasonably quickly.

For each of the time periods of different aggregate request rates, Figure 11 plots the IBT value distribution, and Table 1 shows the measured throughput and probing delay. For the 4 million rps workload, about 90% of the IBT values fall between 650 and 900, and the actual throughput of Maestro is 1.70 million rps, which is the same as the maximum rps achieved with a static IBT of 1200 in the previous experiment. The average probing

Request Rate	Maestro-R-R	NOX	Beacon
4M	1.70M / 2.8ms	1.84M / 342.9ms	2.67M / 10.7ms
1.4M	1.40M / 1.8ms	1.40M / 3.9ms	1.40M / 8.0ms
0.85M	0.85M / 1.4ms	0.85M / 1.6ms	0.85M / 2.1ms

Table 1: Throughput(rps) and probing delay under different request rates(rps)

delay for Maestro is 2.8ms. When the emulator's offered request rate is 1.4 million rps, about 90% of the IBT values fall between 250 and 650. The actual throughput of Maestro is 1.40 million rps, which is the same as the emulator's offered request rate. The average probing delay for Maestro is 1.8ms. When the offered request rate is 0.85 million rps, about 90% of the IBT values fall between 10 and 100, the throughput of Maestro is 0.85 million rps, and the average probing delay is 1.4ms. The long tails in these distributions come from the transition periods from one offered request rate to another, where the IBT needs to be gradually adjusted by the algorithm.

We also evaluate the same scenario using NOX and Beacon, and Table 1 shows the results. When the request rate is 4 million rps, although NOX and Beacon have better throughput, their probing delay performance is much worse than that of Maestro-Round-Robin. When the offered request rate is brought down to 1.4 and 0.85 million rps, where all of the controllers can keep up, we can see that unlike Maestro, NOX and Beacon are not operating at the best batching behavior to keep a low probing delay.

4.4 Throughput and delay scaling

In this section, we conduct experiments that show the throughput and probing delay scaling of all the controllers. We let the emulators generate flow requests at the maximum rate (4 million rps), to stress test all the controllers. We run each experiment five times, using both the 79-switch and the 1347-switch network topologies. In each network topology, we let emulators generate requests with both uniform and skewed rates. The 79-switch with skewed request rates is essentially scenario two in Figure 6, while for the 1347-switch case the rates distribution shape is similar, just with the difference that the requests are from more switches.

Figures 12 and 13 show the throughput scalability with an increasing number of worker threads under the different distributions and topologies. The vertical axis in each figure is the achieved throughput relative to the absolute throughput value at one worker thread. We can see that Maestro-Dynamic-Partition has the best scalability all the way up to seven worker threads, while Maestro-Round-Robin follows in second place. For the 79-uniform workload, Maestro-Static-Partition comes in third ahead of NOX. However, under the 1347-skewed

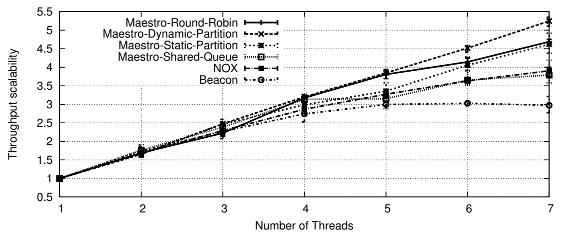


Figure 12: Throughput scalability, 79 - uniform

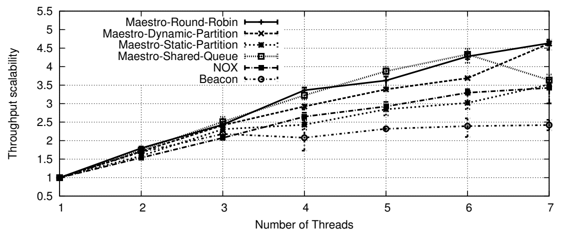


Figure 13: Throughput scalability, 1347 - skewed

workload, this outcome is reversed with NOX ahead of Maestro-Static-Partition. Beacon scales pretty well up to three worker threads, but after that the scalability more or less flattens out. Maestro-Shared-Queue scales well up to six worker threads, but either flattens out in 79-uniform, or becomes worse in 1347-skewed. Because all worker threads have to share the same work queue, its poor scalability is expected. The other two experiments (79-switch with skewed rates, and 1347-switch with uniform rates) show similar throughput scalability results, so we do not include the figures here.

Table 2 shows the absolute throughput values for all controllers using seven worker threads, for the two network topologies and the two request rate distributions. In all cases Maestro-Static-Partition has the best throughput, which we believe is because of its larger maximum read size than the other Maestro designs, and in Maestro the application is executed once to process all requests in one batch instead of being executed multiple times as in NOX and Beacon. Maestro-Dynamic-Partition's throughput follows closely, especially in the experiments with skewed request rate distributions. In these cases, the dynamic re-partitioning can better balance the workload in the worker threads. Note that the better throughput of Maestro-Static-Partition is also because of its larger buffer size, at the cost of increased delay. Also we believe that in more dynamic scenarios where the request rate of switches changes over time, throughput of the Maestro-Static-Partition will be worse. Although the lock synchronization in Maestro-Round-Robin prevents it from achieving the best throughput, it is not too far behind.

Controller	Throughput, request per second			
	79-U	**79-S**	**1347-U**	**1347-S**
Maestro-Round-Robin	2.76M	2.60M	2.41M	2.25M
Maestro-Dynamic-Partition	3.11M	2.94M	2.69M	2.36M
Maestro-Static-Partition	3.43M	3.09M	3.01M	2.46M
Maestro-Shared-Queue	1.25M	1.10M	0.75M	0.74M
NOX	2.51M	2.55M	2.51M	2.41M
Beacon	3.08M	3.02M	2.96M	2.34M

Table 2: Absolute throughput values using seven worker threads, U stands for uniform, S stands for skewed

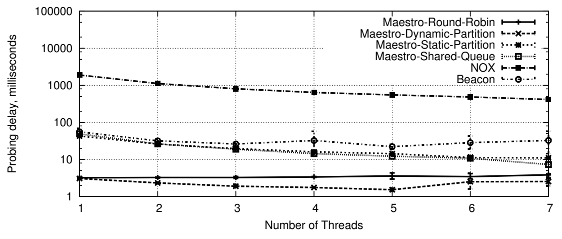


Figure 14: Probing delay in log scale, 79 - uniform

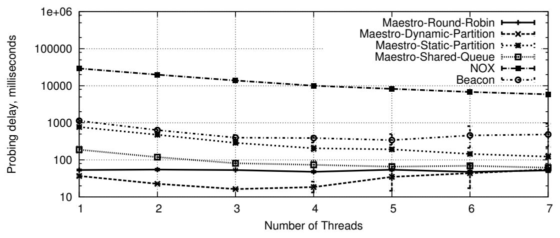


Figure 15: Probing delay in log scale, 1347 - skewed

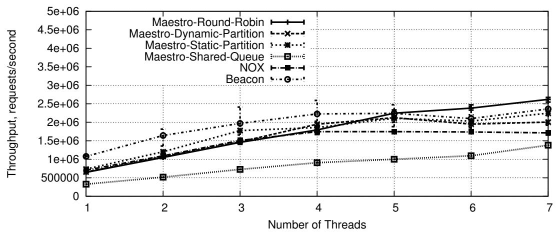


Figure 16: Absolute throughput values, 4 switches

As shown in Figure 14 and Figure 15, the probing delay performance of Maestro-Round-Robin and Maestro-Dynamic-Partition (in which the IBT adaptation algorithm is enabled) are much better than static designs. This is not only because Maestro has much better fairness in throughput allocation, but also because of the IBT adaptation algorithm which prevents the batch from growing too large. Because in Maestro-Shared-Queue worker threads have to synchronize on a shared queue, and because its throughput is much worse, its delay performance is not as good. We believe the very bad probing delay performance of NOX is due to its very large read batching size (512KB). Again, we do not include the figures for the two other experiments (79-switch with skewed rates and 1347-switch with uniform rates) because they show similar results.

4.4.1 Effect of small number of source switches

Instead of having flow requests coming from a larger number of source switches, in this experiment we let the emulators generate flow requests from a small network with only four switches with a total request rate of 4 million rps. This workload is the worst case for any design which assigns source switches to worker threads, because flow requests from one switch can only be processed by one worker thread. Therefore, it is impossible to evenly distribute the requests among more than four worker threads. As shown in Figure 16, Maestro-Round-Robin not only has the best scalability under this workload, but also achieves the best absolute throughput for seven worker threads. The throughput of Maestro-Shared-Queue also keeps growing, although it is still the worst in absolute terms. Throughput of all other systems stops increasing for more than four worker threads.

5 Conclusion

Flexibility and direct control make OpenFlow a popular choice for different networking scenarios today, but the performance of the OpenFlow controller must be optimized not only for raw aggregate throughput, but also to simultaneously achieve fair capacity allocation and low latency. We have systematically evaluated and compared different design choices. The results have shown that the Maestro-Round-Robin design can achieve near optimal fairness in system capacity allocation, while at the same time having throughput scalability second only to Maestro-Dynamic-Partition. The IBT adaptation algorithm of Maestro can effectively adjust the batching behavior dynamically according to the aggregate input rate to control request handling latency, while at the same time achieving high throughput. Simply put, the Maestro-Round-Robin design with the adaptive batching algorithm achieves the best balance between fairness, latency and throughput among all available OpenFlow controller designs today.

Acknowledgements

This research was sponsored by NSF CAREER Award CNS-0448546, NeTS FIND CNS-0721990, NeTS CNS-1018807, by an Alfred P. Sloan Research Fellowship, an IBM Faculty Award, and by Microsoft Corp. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the Alfred P. Sloan Foundation, IBM Corp., Microsoft Corp., or the U.S. government.

References

[1] The Beacon OpenFlow controller. http://www.openflowhub.org/display/Beacon/Beacon+Home.

[2] The Global Environment for Network Innovation. http://www.geni.net/.

[3] Maestro platform. http://code.google.com/p/maestro-platform/.

[4] The Open Networking Foundation. http://www.opennetworkingfoundation.org/.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.

[6] J.R. Ballard, I. Rae, and A. Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. apr 2010.

[7] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, November 2010.

[8] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A system for scalable openflow control. Technical Report 10-11, Rice University, December 2010.

[9] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 2:1–2:6, New York, NY, USA, 2010. ACM.

[10] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *In Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[11] N. Feamster, A. Nayak, H. Kim, R. Clark, Y. Mundada, A. Ramachandran, and M. bin Tariq. Decoupling Policy from Configuration in Campus and Enterprise Networks. 2010.

[12] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[13] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *ACM Computer Communication Review*, July 2008. Short technical Notes.

[14] S.W. Han, N. Kim, and J.W. Kim. Designing a virtualized testbed for dynamic multimedia service composition. In *Proceedings of the 4th International Conference on Future Internet Technologies*, pages 1–4. ACM, 2009.

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, R. Ramanathan M. Zhu, Y. Iwata, H. Inoue, T. Hama, , and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proc. Operating Systems Design and Implementation*, pages 351–364, October 2010.

[16] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM Computer Communication Review*, 38:69–74, April 2009.

[17] J. Naous, R. Stutsman, D. Mazières, N. McKeown, and N. Zeldovich. Delegating network security with more information. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 19–26. ACM, 2009.

[18] A.K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 11–18. ACM, 2009.

[19] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. *ACM SIGCOMM Computer Communication Review*, 39(4):39–50, 2009.

[20] R. Sherwood, M. Chan, A. Covington, G. Gibb, M. Flajslik, N. Handigol, T.Y. Huang, P. Kazemian, M. Kobayashi, J. Naous, et al. Carving research slices out of your production networks with OpenFlow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[21] Neil Spring, Ratul Mahajan, and David Wetheral. Measuring ISP topologies with RocketFuel. In *Proc. ACM SIGCOMM*, August 2002.

[22] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *Eighth ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[23] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In *Passive and Active Measurement*, pages 201–210. Springer, 2010.

[24] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *INM/WREN*, 2010.

[25] K.K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown. The stanford openroads deployment. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, pages 59–66. ACM, 2009.

[26] M. Yu, J. Rexford, M.J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proc. ACM SIGCOMM*, August 2010.