Maestro: A System for Scalable OpenFlow Control

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 initial establishment of every flow by contacting related switches. Thus the performance of the controller could be a bottleneck. This paper shows how this fundamental problem is addressed by parallelism. The state of the art OpenFlow controller, called NOX, achieves a simple programming model for control function development by having a single-threaded event-loop. Yet NOX has not considered exploiting parallelism. We propose Maestro which keeps the simple programming model for programmers, and exploits parallelism in every corner together with additional throughput optimization techniques. We experimentally show that the throughput of Maestro can achieve near linear scalability on an eight core server machine.

***Keywords*-OpenFlow, network management, multithreading, performance optimization**

I. INTRODUCTION

Inspired by the 4D architecture [6], the emerging OpenFlow [9] switch architecture separates the two main functions of a classical router/switch: data plane packet switching and control plane routing decision making. The OpenFlow switch devices only implement the data plane packet switching functionality. The OpenFlow controller machine takes charge of the control plane functionality by installing and deleting flow entries on switch devices. OpenFlow creates new opportunities to realize rich networking functions, by allowing the users to flexibly program control plane functionalities on the OpenFlow controller, and to freely control the data plane of the switch devices. The success of OpenFlow can be seen from the large number of recent use cases: programmable network testbeds [8][19][13], datacenter network designs [15][12][1], enterprise network designs [11][10][5], network measurement systems [2][17], to name just a few recent examples.

One fundamental feature of OpenFlow is that, the controller is responsible for establishing every 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 (usually implemented in TCAM) to match this flow, the first packet will be forwarded to the controller. We call this first packet a "flow request". The controller will first check this flow against security policies to see whether it should be allowed, and if allowed the controller needs to compute a path for this flow, and install flow entries on every switch along the chosen path. Finally, the packet itself will be sent back to the origin switch from the controller. As the network scales in size, so will the number of flows that need to be established by this process. If the controller does not have the capacity for handling all these flow establishment requests, it will become a network bottleneck.

With OpenFlow switches already being used for designing large-scale datacenter networks connecting hundreds of thousands of servers, improving the performance of the controller system to keep up with the rising demand becomes a critical challenge. Measurements of traffic from data centers with different sizes and purposes [3] have shown that the number of concurrent active flows is small, which implies that OpenFlow switches can be a well fit for being applied in building data center networks. However, the authors show that for a data center which has 100 edge switches, the centralized controller can expect to see about 10 million flow requests per second. This creates a fundamental challenge for the centralized OpenFlow controller to be deployed in a larce scale data center.

Fortunately, such flow request processing in the controller is potentially parallelizable. There are no complicated data dependencies, all the computation is about checking a flow against security policies, finding a path for it, and sending out flow entry configuration messages. With the commoditization and wide-spread adoption of multi-core processors (AMD has already shipped x86 processors with 12 cores), the time is right to examine how to best address the controller bottleneck problem by exploiting parallelism.

NOX [7], the state-of-the-art OpenFlow controller system, is a centralized and single-threaded system. Although this design achieves a simple programming model for control function development by having a single-threaded event-loop, it cannot take advantage of current advances in multi-core technology. We find that another inefficiency of NOX is that each flow request is processed individually, and all packets generated accordingly are sent individually. By profiling NOX, we find that about 80% of the flow request processing time is spent in sending out messages individually. Through a microbenchmark experiment presented later on in the design section, we show that the overhead of multiple socket write operations to send each packet individually to the same destination instead of a single batched send is very high. Thus, besides investigating the use of multi-threading to best leverage the capability of multi-core processors, another important principle we focus on is correctly using batching to improve the efficiency of the system.

In this paper, we present a new controller design called Maestro, which keeps a simple single-threaded programming

model for application programmers of the system, yet enables and manages parallelism as a service to application programmers. It exploits parallelism in every corner together with additional throughput optimization techniques to scale the throughput of the system. We have implemented Maestro in Java. We experimentally show that the throughput of Maestro can achieve near linear scalability on an eight core server machine.

While the individual design decisions, optimization techniques, and fine-tuning techniques we employ are what make Maestro unique, the most important value of Maestro lies in the fact that it is a complete system engineered to meet the specific characteristics and needs of OpenFlow, that it addresses a real-world open challenge, and that it will have immediate positive impact on many deployed and to be deployed OpenFlow networks as Maestro will be open-sourced in the very near future.

The rest of this paper is organized as follows. In Section II, we present the design and implementation of Maestro. In Section III, we experimentally evaluate the performance of Maestro to show the benefits of our techniques. We discuss the related work in Section IV, and conclude in Section V.

II. System Design and Implementation

A. The Overall Structure of Maestro

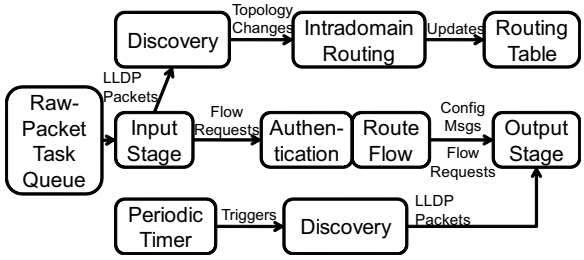

Fig. 1. The overall structure of Maestro.

Functionality Structure

Figure 1 shows the functionality structure of Maestro. Any OpenFlow controller will share these similar functionalities, although it may implement them in different ways. Maestro sends and receives OpenFlow messages to and from network switches via per switch TCP connections. The "Input Stage" and "Output Stage" handle low level details of reading from and writing to socket buffers, and translating raw OpenFlow messages into and from high level data structures. These low level functionalities stay fixed with a particular version of the OpenFlow protocol. Other high level functionalities may vary and are implemented in modules called "applications" in Maestro. Programmers can flexibly modify the behavior of these applications, or add new applications to meet different goals. In Figure 1, the applications are: "Discovery", "IntradomainRouting", "Authentication" and "RouteFlow".

When switches join the network by setting up TCP connections with Maestro, the "Discovery" application periodically sends out probing messages to the neighbors of each switch. The probing messages conform to the vendor-neutral Link Layer Discovery Protocol (LLDP). This is represented by the application execution path at the bottom of the figure. Then when "Discovery" receives bounced back probing LLDP messages, it knows in the message that from which origin switch this packet is sent, thus it can discover the topology of the network. Such extra discovery process is mandatory because in the current version of OpenFlow, version 1.0.0, switches do not discover neighbors themselves. When "Discovery" finds topology changes, it initiates the "IntradomainRouting" application to update the "RoutingTable" data structure accordingly. This is represented by the application execution path at the top of the figure. This "RoutingTable" contains the all-pair-shortest paths for the entire network, and will be used by the "RouteFlow" application to find paths for flow requests.

When Maestro receives a flow request from a switch, this request will be first checked against security policies in the "Authentication" application. Only when allowed, the "RouteFlow" application will try to find a path for this request, and generate one flow configuration message for each of the switches along the chosen path. From now on we call these two applications together the "flow process stage". After that, all flow configuration messages will be sent to their destination switches, and the flow request packet itself will be sent back to the origin switch. This is represented by the application execution path in the middle of the figure, which we call the "flow request execution path" from now on.

All three application execution paths run in parallel. Maestro makes sure that if the "RoutingTable" data structure gets updated while the "RouteFlow" application is already running, "RouteFlow" continues with the older version of "RoutingTable" to generate consistent results. Next time when "RouteFlow" executes it will start to use the latest "RoutingTable".

We choose Java to be the programming language for Maestro, and there are several reasons. First, Java programs are considered to be easy to write and to maintain. Java programs are more secure, so it is relatively more easy to debug and to maintain. Also Java can support dynamic loading of applications and data structures without recompiling and restarting the whole system more easily, so it will make Maestro very flexible to extend. Second, it is very easy to migrate Java code to different platforms as long as there is Java Virtual Machine support on that platform. Usually the code needs very little or even no modification to work on another platform, which makes Maestro more flexible. Third, although Java is considered to be less efficient than C or C++, but we argue and show by evaluation that, Maestro can achieve overall good performance and scalability by incorporating the right design and optimization techniques.

Multi-Threading Structure

Maestro has a task manager which provides a unified interface for managing pending computations. Any computation can be wrapped into a "task" java class and be submitted to the task manager. The task manager manages a number of running worker threads to execute these submitted tasks. The actual

number of worker threads is chosen based on the number of processor cores in the controller machine. Take the flow request execution path for example, which is also the focus of the parallelization in Maestro. OpenFlow raw packets received from the sockets are wrapped into tasks together with the input stage code, and put in the raw-packet task queue. Any available worker thread will pull one task from this queue, and execute the input stage code to process the raw packet in that task. At the end of the input stage, there will be flow requests generated to be processed by the "Authentication" and "RouteFlow" applications, i.e. the flow request stage. These requests are wrapped into tasks together with the application code, and put in the dedicated task queue of this worker thread. At the end of the flow request stage, the generated configuration messages and the flow requests themselves will be wrapped into tasks together with the output stage code, and again put in the dedicated task queue. Finally the worker thread will execute these output stage tasks to send the messages out to their destination switches. Maestro enables multiple instances of the flow request execution path to be run concurrently by different worker threads. Each application remains simple and single-threaded. More details about why we choose such a design will be provided later.

One can also choose to leave the burden of multi-threading to programmers of the high level applications. For example, in the "Authentication" and "RouteFlow" applications, programmers can divide the flow requests into several parts, and create a number of java threads to work on these parts concurrently. However, we argue that this approach is not efficient. This is because high level application programmers can only parallelize the flow process stage; the low level input and output stages still need to be parallelized by the underlying controller platform. With threads created and managed by the high level applications and threads created and managed by the controller platform co-existing, scheduling threads optimally becomes much more difficult. This is also because it is relatively hard for programmers when writing the applications to get and consider runtime platform conditions, such as number of cores available, free memory available, etc.

As a result, in our design, Maestro keeps a simple programming model for programmers. Application programmers do not need to deal with multi-threading within applications, they simply write single-threaded applications. By composing and configuring the applications, the Maestro system can run multiple instances of an application execution path concurrently to achieve parallelism, with one worker thread handling one instance.

B. Multi-Threading Design

Design Goals

1) Distribute work evenly among available threads/cores, so that there will be no unbalanced situation where some thread/core has too much work to do while others are idling.
2) Minimize the overhead introduced by cross-core and cache synchronization.
3) Minimize the memory consumption of the system to achieve better memory efficiency.

Distribute Work Evenly

To maximize the throughput of the system, work has to be evenly distributed among worker threads in Maestro, so that no processor cores sit idle while there is pending work to do. At first glance, one may choose to design the system in a way that incoming flow requests are evenly divided and directly pushed to all the dedicated task queues of the worker threads, in hope of achieving an even workload distribution. However, in the case of OpenFlow, this solution does not work, because each flow request can require a different number of CPU cycles to process. For example, a flow with a longer forwarding path leads to more configuration messages generated, so there is more data to process and to send. Although the path length can be looked-up given access to the all pair shortest paths, this will introduce an additional look-up overhead in the low level raw-packet processing stage, and violate the modular abstraction boundary because the all pair shortest paths are maintained by programmer applications and not supposed to be exposed to the low level system. In addition, security authentication performed on flow requests can also require a varying number of CPU cycles, depending on what security policy governs that flow. This is much more difficult to predict than the length of the path. In summary, different OpenFlow flow requests can require significantly differing numbers of CPU cycles to process.

As a result, we design the task manager in a "pull" by worker threads fashion, instead of actively pushing work to worker threads. All worker threads share the same raw-packet task queue, so that when there is a pending raw-packet task, any available worker thread can pick it up. Although in this way worker threads have to synchronize on this raw-packet task queue, because task queue operations are very lightweight, and the time spent in contending for locks is much smaller than the time spent in handling the entire request, the overhead of queue synchronization is relatively small. In the evaluation section we will show that, the synchronization overhead introduced by this pull design is negligible. The benefit of this design is that work is evenly distributed among worker threads.

Minimize Cross-Core Overhead

When running code or actively used data is moved from one processor core to another, especially from one processor to another processor which do not share the same cache, there is overhead in synchronizing the core state and the cache. In Maestro we want to minimize this kind of cross-core overhead to maximize the performance of the system.

First of all, we use the `taskset` system call to bind each worker thread to a particular processor core. By doing this, we can prevent active running code from being rescheduled to another processor core by the operating system, which will introduce very large overhead. We call this design the "core-binding".

Second, one way to minimize cache synchronization is to make sure that all computation for processing one flow

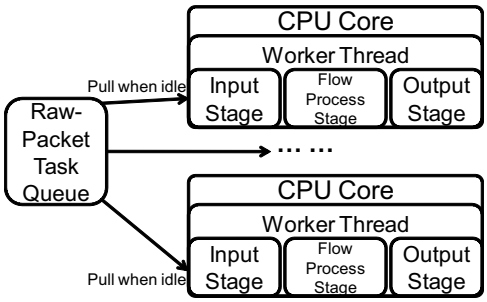

Fig. 2. Pull-based work distribution and thread & core binding design.

request is done by the same processor core. By doing this, data generated and used during the whole process will not have to be moved from the cache of one core to another one. To achieve this, the task manager framework is designed as shown in Figure 2. For input stage tasks, all worker threads still share the same raw-packet task queue, so that they can pick up pending raw packets when they are idle. In addition, each worker thread has its own dedicated task queue. All the tasks for the flow process stage and output stage will be submitted to this dedicated task queue of the worker thread in which the input stage is executed. A worker thread always tries to pull tasks from its own dedicated task queue first, and only when that queue is empty does it access the shared raw-packet task queue. This design ensures not only that work load is evenly distributed among worker threads, but also that all computations related to one flow request are processed by the same worker thread. We call this design the "thread-binding"

In Section III-E of the evaluation, we quantitively show that both the "core-binding" and the "thread-binding" designs are critical to scale up the througput of the system.

Minimize Memory Consumption

When the rate of incoming flow requests exceeds the throughput of the system, data such as raw packets, flow requests generated by the input stage, and configuration messages generated by the output state will accumulate. If too much data is pending in queues to be processed, it will not only consume a lot of memory, but also greatly hurt the system performance. This is because in general more memory used means worse locality for the memory hierarchy, and exhausting the memory could have disastrous performance consequences. Moreover, because Maestro is implemented in Java, the overhead of garbage collection becomes very high when a large amount of memory is used, thus memory allocation will become extremely slow also. The ultimate goal is to minimize the memory consumption, while making sure the system has enough buffered data to be processed. The latter part is also very important. If not enough raw packets are buffered, when worker threads drain all pending raw packets including those within the socket buffers, it will take some time for TCP to re-grow the congestion window and feed enough requests again to Maestro, especially when the socket buffer is small and the network latency is large. This could potentially lead to CPU idling.

Thus, thresholds must be carefully configured in the system to keep any queue from growing too large but remaining at a reasonable size. However, having one threshold for each queue is not a good design, because there will be many thresholds to tune. Whenever some conditions change significantly (e.g. a network latency change), these thresholds have to be adjusted. The more thresholds there are, the more difficult it is to come up with a good configuration. The solution to this problem is to make sure that there is no data accumulated in queues between the input and flow process stage, and between the flow process stage and the output stage. To achieve this, we employ strict priority in scheduling tasks to worker threads, and we give different priorities to the tasks for the different stages. Tasks for the output stage receive high priority, tasks for the flow process stage receive medium priority, and tasks for the input stage naturally receive low priority because they are in the shared raw-packet task queue. As a result, the only queue that can become backlogged is the raw-packet task queue. In order to minimize memory consumption, there only needs to be one threshold configured: how many pending raw packets are allowed in the raw-packet task queue to be processed by the input stage, which we call the "pending raw-packet threshold", or PRT in short. In addition, we have a dedicated thread that receives incoming raw packets from the socket buffers. Since receiving from sockets is very lightweight compared to flow request processing, this dedicated thread can keep the queue filled to the threshold without requiring many CPU cycles.

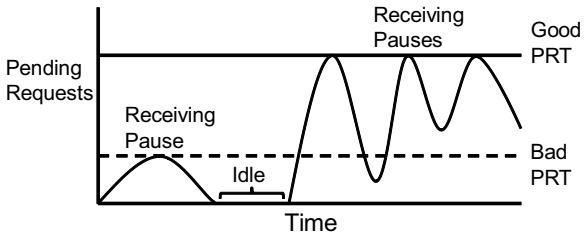

Fig. 3. Good PRT vs bad PRT.

In Maestro, when there are more pending raw-packet tasks than the PRT, the receiving thread is paused. When there are fewer requests than the PRT, the receiving thread is resumed. The PRT could and should be calibrated before an initial deployment of Maestro, and again when network conditions such as the topology and the network latency change significantly. Usually a network with a higher request input rate and a higher latency requires a larger PRT, and vice versa. This calibration can be achieved by injecting flow requests from a network emulator at a rate higher than the system throughput to Maestro. The use of such a high rate is important because the calibrated PRT will guarantee that there will be enough room for any request rate while achieving high memory efficiency. Figure 3 shows the difference between a correct and an incorrect PRT. If the PRT is chosen to be too small, after the receiving thread pauses, the worker threads could drain the raw-packet tasks so fast that they will become idle even before new requests are received because of the TCP

ramp-up delay. A good choice of PRT is the minimum value that can make sure that under the maximum input rate there is no case when the raw-packet task queue is ever completely drained. In the evaluation section we will use our emulation tool to calibrate the PRT for Maestro.

C. Batching

There is unavoidable overhead associated with each of the three stages in the flow request execution path. For example, Maestro needs to start an instance of the "Authentication" and "RouteFlow" applications, and needs to pass data between any two stages and also between two applications. Another example is the overhead of `socket send()` system calls. When the incoming rate of flow requests is very high, any of the overheads mentioned above could become significant. As a result, we propose to use batching mechanisms to minimize such overhead.

Input Batching

For each worker thread, there is one input batching queue in the input stage to temporarily hold the flow requests that it generated from the raw packets. This input batching queue enables the input stage to create a flow process stage task that can process multiple flow requests in one single execution. The two applications of the flow process stage are implemented to be aware of the input batching queue. They should process all the pending flow requests in the input batching queue in one batch during one execution. However, if we allow the input batching queue to grow very large, it will consume a lot memory and increase the average delay experienced by all flow requests. Thus, there is a tunable threshold which decides when the input stage should create a flow process stage task that processes all flow requests in the input batching queue. We call this threshold the "input batching threshold", or IBT for short.

When the rate of incoming flow requests is lower than the throughput of the system, from time to time worker threads will have no pending task to run. We still keep the same IBT, but whenever a worker thread finds that the raw-packet task queue is about to be drained out, it will force the input stage to create a flow request stage task with as many requests as there are currently in the input batching queue, ignoring the IBT. Through this simple design, we can achieve dynamic batching adjustment that responds to the changing load of the system, so that the average processing delay is kept low, without having to develop a threshold adjustment algorithm.

The IBT could and should be calibrated once at initial deployment and when network conditions such as the topology and the network latency change significantly. This calibration is done under a rate of incoming flow requests that is larger than the throughput of the system. The same value will work for a lower rate as discussed in the previous paragraph. During the calibration, a value is chosen where the average delay is minimized while the throughput is maximized.

Output Batching

Because for each `socket send()` call there are both fixed and variable, per-byte overheads, when there are multiple messages to be sent to the same destination, sending them all in one `socket send()` call can be much less expensive than sending each of them individually. We conduct a microbenchmark experiment to demonstrate this, the result is shown in Figure 4.

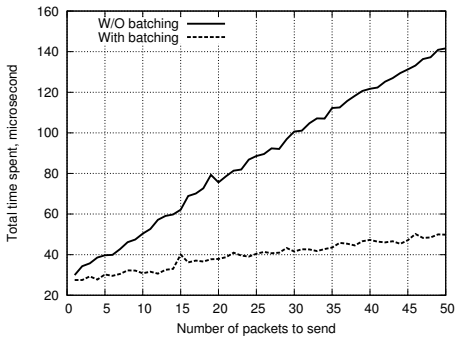

Fig. 4. Microbenchmark for output batching.

In this microbenchmark, we vary the number of 100-byte messages (a typical size for OpenFlow messages) to send to the same destination from 1 to 50. In the first experiment, we send each of them individually, and in the second experiment we send all of them together with one `socket send()` call. We run each experiment 100 times and measure the average time spent in each run. As shown in the figure, the time for sending all messages together grows much slower than that for sending them individually.

To reduce this overhead in Maestro, we perform output batching. When the flow process stage processes one batch of flow requests, and generates a set of messages that need to be sent, we first group these messages by their destinations. Then, all the messages for the same destination are sent together in one single `socket send()` call. If there are too many bytes to send that it cannot be done with only one call, we will try multiple calls. For each call we will send as many as possible, which is determined by the socket's then available buffer space. In addition, because only one thread is allowed to call the `socket send()` on one socket at a time, we add the following feature to further minimize the wait time. When a worker thread tries to call `socket send()` on one socket but finds out another worker thread is already locking that socket, instead of waiting, the thread will process other pending output stage tasks in its dedicated task queue in a round-robin fashion, until it finds one socket that is not being locked. This solution greatly improves the output efficiency.

III. Evaluation

In this section, we evaluate Maestro through a number of different scenarios, to show how our design choices affect the system's performance. Also, we compare Maestro against NOX (version 0.8.0 full beta, checked out on Jul-10-2010, compiled without the ndebug option enabled) to see how much benefit we get from addressing NOX's limitations. Because the design of the security policy data structure and checking algorithm is not the focus of this paper, in the evaluation

we use a simple policy that allows all flow requests in the "Authentication" application in Maestro, and also in the corresponding application in NOX.

A. Experiment Setup and Methodology

We implement a network emulator which can emulate all OpenFlow switch control plane functionalities and can generate flow requests at different rates. The emulator also forwards control plane LLDP packets to help the controller discover the network topology. It also delivers all flow configuration messages to the emulated target switches.

We run Maestro on a server machine with two Quad-Core AMD Opteron 2393 processors (8 cores in total) with 16GB of memory. Because there is one dedicated thread for receiving requests, and there are several other threads responsible for managing the Java virtual machine (such as class management and garbage collection), we dedicate one processor core for such functionalities, while the remaining 7 cores are used by worker threads. Thus the best throughput is achieved with 7 worker threads on this 8 core server machine. This machine has four PCI-E 1Gbps NICs to provide enough network bandwidth. At least 3Gbps network bandwidth is necessary because for each request the controller not only needs to generate several (depending on the length of the chosen path) flow configuration messages for switches along the path, but also needs to bounce the flow request packet itself back to its origin. So when Maestro achieves its maximum throughput, it needs to send out data at a rate of around 2.5Gbps. The machine is running Ubuntu 9.04 with a 2.6.31 kernel, and is configured with a 64-bit 1.6.0_19 JDK.

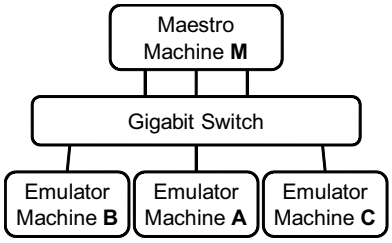

Fig. 5. Experiment emulator setup.

Because for the emulator we only have machines with one 1Gbps NIC, we need at least three emulator machines to be able to test the maximum throughput of Maestro. Thus we run the emulator distributedly on three machines as shown in Figure 5. There is one master machine A and two slave machines B and C, and each of them is connected to a gigabit Ethernet switch by a 1Gbps link. Maestro is running on the server machine M, and has three 1Gbps links to the gigabit switch. Each of the emulator machines emulates one third of the OpenFlow switches in the emulated network, and each of them has one separate 1Gbps path, A-M, B-M and C-M, to reach M. Flow requests generated on them will be sent to Maestro, and configuration messages will be sent from Maestro to their target machines, all along these three paths. In addition each slave machine also has one path to the master machine A, B-A and C-A. When LLDP packets need to be sent from one switch to its neighbor switch on another emulator machine, it will be forwarded, via the master machine A if necessary. For example, assuming switch X is on B and its neighbor switch Y is on C, the LLDP packet from X to Y will first reach A and then get forwarded to C by A. Although B-A and C-A share bandwidth with B-M and C-M respectively, because the LLDP traffic is very light, Maestro's throughput will not be affected.

To provide a common basis for direct comparisons between Maestro and NOX, we run experiments on a common 79-switch topology [14]. For each emulated switch, the emulator emulates one end host connected to it. We generate fifteen million requests between randomly chosen source and destination end host pairs to get aggregate performance measurements. All the requests are generated at uniform rate without bursty behavior.

We use two metrics for measuring the performance. The first one is Maestro's throughput in requests per second (rps), for which a larger value is better. The second one is the delay experienced by the flow requests, for which a smaller value is better. This delay is the end-to-end delay measured by the emulator. The emulator creates a request, records the starting timestamp, and calls the blocking socket send function. When the emulator receives the request itself bounced back from Maestro, it calculates the end-to-end delay using the starting timestamp.

B. Calibrating the PRT

When calibrating the PRT value to use for a network, we let the emulator generate flow requests as fast as possible, regulated only by the blocking TCP send call. The emulated network environment which we use has a small latency, and can send requests at a rate larger than the maximum throughput Maestro can achieve. Together with measuring the throughput, we also measure how many times during an experiment there are zero pending requests, which means potential idling for the worker threads. A good choice of the PRT value for a particular network is the smallest value that by using it we can achieve the best throughput, and have no situation where there are zero pending requests left while the switches are still trying to send more.

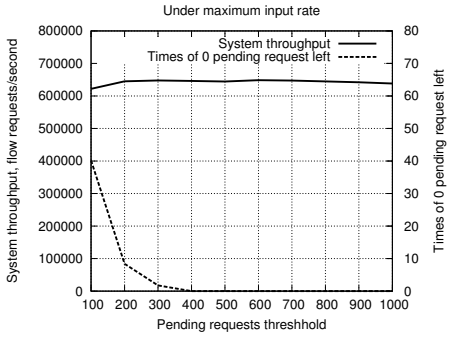

Fig. 6. PRT calibration under maximum load.

As shown in Figure 6, allowing only 100 pending requests is not a reasonable choice. This is because even for the emulated network which has very small latency, there are still many times that the TCP connections cannot ramp up quickly enough after a pause, so there are zero pending requests left in the system. This leads to suboptimal throughput. From the results in this figure, we choose a PRT value of 400 for our emulated network because the one case of zero pending requests happens at the end of the emulation when the emulator stops sending more requests.

C. Calibrating the IBT

When calibrating the IBT value to use, we again let the emulator generate flow requests as fast as possible. For the variable IBT value, first we choose 1, 2, 4, ..., 2048, 4096. This helps us find the right range to zoom in on relatively quickly.

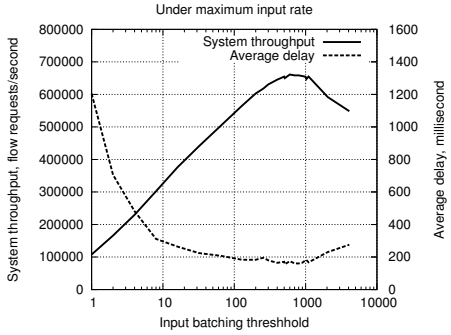

Fig. 7. IBT calibration under maximum load. Note that delay will be much smaller under normal load.

Figure 7 shows the result. In this figure, the solid line stands for the throughput of the system, for which larger is better. The dashed line stands for the average delay experienced by all the flow requests, for which smaller is better. When the IBT value is 1, which is effectively no input batching at all, even with 7 worker threads the throughput of the system is only around 100,000rps, and the average delay is as large as 1.2s. When we grow the IBT value, the throughput gets larger and the delay gets lower, until around 512 where we get the lowest delay and the best throughput. Notice that the delay measured here is under the maximum input rate. Usually in a network without such heavy load, the average delay is much smaller as we will show later. Here we optimize for better throughput, so we zoom into the region around 512, and through additional finer-grained calibration we find out that any IBT value between 510 and 530 will work equally well.

D. Comparing Maestro and NOX

We use the same emulated network to evaluate NOX. We find out that, under maximum load, NOX can achieve a maximum throughput of 21,000rps, and the average delay is 4.11s. The delay is this large because the controller machine has 79 TCP connections with the 79 switches, and flow requests are generated from randomly chosen switches. Thus there are 79 socket write buffers on the switches and 79 socket read buffers on the controller machine to buffer flow request packets. Since each flow request packet is 80 bytes, on average each buffer only needs to have about 21,000*4.11*80/(2*79) = 43.7KB of pending packets to build up such a large delay. Because in the operating system that we use, the size of the socket write buffer starts at 16,384 bytes, the size of the socket read buffer starts at 87,380 bytes, and both socket write and socket read buffers are dynamically adjusted by TCP, the average 43.7KB socket buffer occupancy value is within expectation. The poor performance is the result of the fact that NOX can only utilize one core, and does not have the additional features of Maestro such as input and output batching, core and thread binding, etc.

We also run Maestro under the same feature settings to provide a direct comparison with NOX. That is, Maestro is restricted to run with only one thread, and with the batching and core binding features disabled. In this restricted case, Maestro can achieve a maximum throughput of 20,500rps, and the average delay is 4.27s. There is little surprise here – under the same restricted feature settings, Maestro and NOX have similar performance. It is when running with 7 worker threads and the batching and binding features enabled that Maestro can show much better performance.

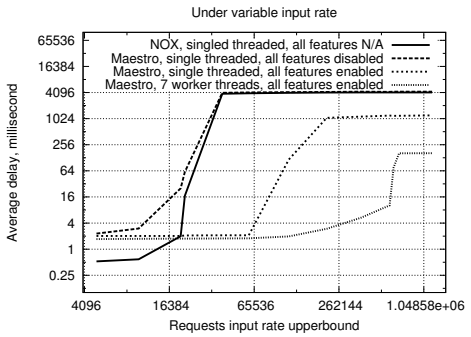

Fig. 8. Average delay under various input rate upperbounds.

We conduct a set of experiments to evaluate how NOX and Maestro perform under different request input rates. We adjust a request input rate upperbound variable, α, in the emulator; the emulator can generate at most α requests per second. If α is larger than the maximum throughput of the controller, TCP will slow down and adjust the real input rate to be equal to the maximum throughput of the controller, and then even if α keeps growing the observed average delay of flow requests will stay the same.

Figure 8 shows the average delay comparison between NOX and Maestro with different feature settings. Because the difference for both the X and Y axis is very large, we grow the request input rate upper bound exponentially, and plot the figure in log scale. When the input rate is under 20,000rps, the average delay of NOX is smaller than 2ms. When the input rate reaches 21,126rps which is about the maximum throughput of NOX, the average delay is 17ms. When the request input rate upper bound keeps growing, it exceeds the maximum

throughput of NOX, thus packets start to accumulate, and TCP slows down. So thereafter the average delay of NOX stays at around 4.11s.

When Maestro is running with similar settings as NOX, that is, single threaded and with the batching and binding features disabled, performance of Maestro is close to but slightly worse than NOX. But if we enable the batching and binding features, even with only one thread Maestro can already show very good improvement. Under light load, average delay is only around 2ms. The maximum throughput is raised to 141,117rps, and when the request input rate upper bound exceeds this limit, the average delay of Maestro stays at around 1.21s.

When Maestro is running with 7 worker threads and the batching and binding features are enabled, because the maximum throughput achieved is as high as 633,290rps, only near the end of this figure the request input rate upper bound exceeds the maximum throughput, and the average delay stays at around 163ms thereafter. When the input rate is 630,000rps which is around the maximum throughput, the average delay is about 76ms. Under light load, the average delay is around 2ms.

Notice that at very low request input rate, the delay of Maestro is slightly larger than NOX. This is because even though the throughput of Maestro is much better, the overhead in processing each individual packet is larger. This extra delay comes from the overhead of the extra steps in Maestro such as task pulling and data passing, and is also because Java is in general slower than C++ in which NOX is implemented. But we argue that these extra steps are worthwhile, because they make it possible to achieve both a simple programming model and very good throughput by exploiting parallelism. This experiment shows that Maestro can achieve a much higher throughput than NOX, and when under a load smaller than the maximum throughput Maestro can process each request at a small delay.

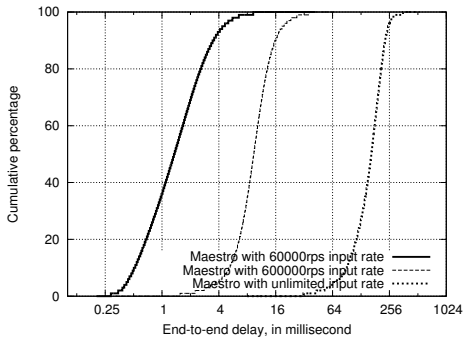

Fig. 9. CDF of delay under different request input rate upperbounds.

In addition, Figure 9 shows the CDF of the end-to-end delay of all flow requests, when Maestro is under different request input rates. The delay distribution of Maestro is relatively stable, especially under lighter load. Even under the maximum request input rate, the worst case delay is smaller than 540ms.

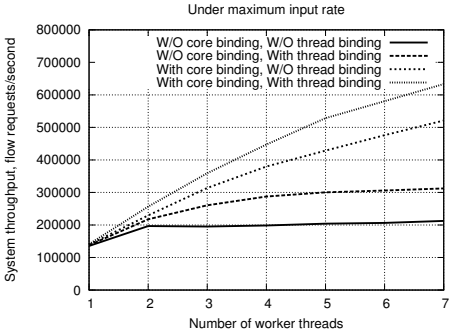

Fig. 10. Controller throughput.

E. Effectiveness of Multi-Threading

Figure 10 shows the throughput scalability of Maestro with different numbers of worker threads. Because the best throughput is achieved with 7 worker threads on the 8 core server machine, we only show the number of worker threads up to 7. As the two "W/O core binding" cases shown in the figure, the feature of binding a worker thread to a core is critical because the overhead to move running code and all data it used among different cores is very expensive. Without such a feature the system cannot scale up very well. If this feature is enabled, as the two upper lines show, the throughput can scale almost linearly with more worker threads in the system. Finally, if the feature of binding a request to a worker thread is also enabled, Maestro shows the best throughput because this feature helps reduce cache synchronization overhead because the whole process of handling one request stays within one worker thread, thus within one processor core.

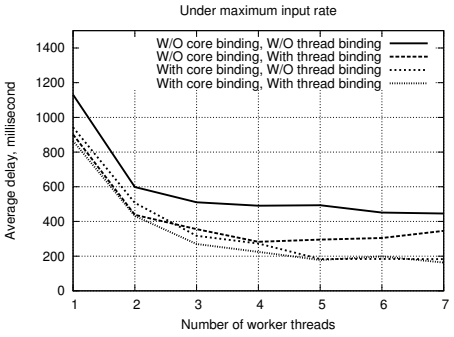

Fig. 11. Average delay under maximum request input rate. Note that delay will be much smaller under normal load.

Figure 11 shows the average delay with different numbers of worker threads. With both of the two features enabled, Maestro shows the best performance in reducing the average delay experienced by all requests. Again notice that the minimum delay of 163ms shown in this figure is for when Maestro is under maximum input rate, and the delay will be much smaller when the load is not as heavy.

In addition, to evaluate the overhead of sharing one raw-packet task queue because of the pull-style work distribution, we measure both the time spent in waiting for acquiring the

lock on the raw-packet task queue, and the time spent in running all tasks, when Maestro is at maximum throughput. It turns out that the time spent in waiting for acquiring the lock is only 0.098% of the time spent in running tasks. This confirms our argument that the overhead of sharing one raw-packet task queue is negligible.

F. Effectiveness of Output Batching

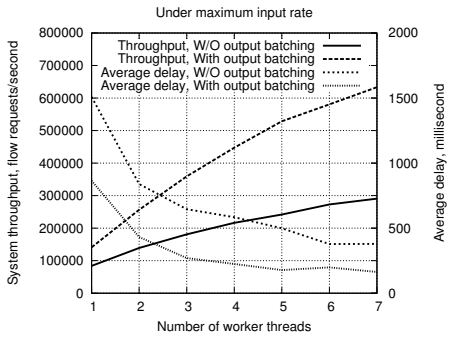

Fig. 12. Performance comparison between whether output batching is enabled.

Figure 12 shows the performance comparison for Maestro with and without the output batching feature enabled, under the maximum rate of input requests. As shown in the figure, when output batching is enabled, both the throughput and the average delay performance is much better than that of when output batching is disabled, due to the reduction in the overhead of sending out messages. Thus output batching is also very critical in improving the performance of Maestro.

G. Discussion

In this section we have shown that, by excessively exploiting parallelism within a single server machine, and additional optimization techniques such as minimizing cross-core overhead and batching, Maestro can achieve near linear scalability in throughput of processing flow requests on a eight-core server machine. Although the largest throughput that Maestro can achieve (600000rps) is still far away from the requirement imposed by a large scale data center (more than 10 million rps), we totally expect that Maestro can be distributed as done similiarly by NOX [16] to scale to tens of millions of flow requests per second. Moreover, the scalability of Maestro within one single multi-core server machine can help cut the number of required distributed controllers by at least ten times of that required by NOX.

IV. Related Work

Ethane [4], the predecessor of NOX and OpenFlow, is an early flow-based networking technology for creating secure enterprise networks. Ethane shows that by restricting reachability in the network before an identity is authenticated by a central controller, strong security policies can be enforced in the network. OpenFlow is a generalization of Ethane, and NOX is a more full-fledged and programmable controller design. However, neither Ethane nor NOX considers exploiting parallelism in their designs.

HyperFlow [18], like Maestro, aims at improving the performance of the OpenFlow control plane. However, HyperFlow takes a completely different approach by extending NOX to 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. The techniques employed by HyperFlow are orthogonal to the design of the controller platform, thus, they can also enable Maestro to become fully distributed to attain even higher overall scalability.

DIFANE [20] 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: Functionalities that can be realized by DIFANE can be off-loaded to switches, while functionalities that require central controller processing can be handled efficiently by Maestro.

V. Summary

Flexibility and direct control make OpenFlow a popular choice for different networking scenarios today. But the performance of the OpenFlow controller could be a bottleneck in larger networks. Maestro is the first OpenFlow controller system that exploits parallelism to achieve near linear performance scalability on multi-core processors. In Maestro, programmers can change the control plane functionality by writing simple single-threaded programs. Maestro incorporates a set of designs and techniques that address the specific requirements of OpenFlow, exploits parallelism on behalf of the programmers, and achieves massive performance improvement over the state-of-the-art alternative. This performance improvement will have a significant positive impact on many deployed and to be deployed OpenFlow networks.

References

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

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

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

[4] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications, pages 1–12, New York, NY, USA, 2007. ACM.

[5] 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.

[6] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. ACM Computer Communication Review, October 2005.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. Applying nox to the datacenter. In Proc. HotNets, October 2009.

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

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

[19] 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.

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