OpenFlow based Flow Level Bandwidth Provisioning for CICQ Switches

Hao Jin, Deng Pan, Jason Liu, and Niki Pissinou
Florida International University

***Abstract*—Flow level bandwidth provisioning offers fine granularity bandwidth assurance for individual flows. It is especially important for virtual network based experiment environments, to isolate traffic of different experiments or different types, which may be fed to the same switch or router port. Existing flow level bandwidth provisioning solutions suffer from a number of drawbacks, including high implementation complexity, poor performance guarantees, and inefficiency to process variable length packets. In this paper, we study flow level bandwidth provisioning for combined-input-crosspoint-queued switches in the OpenFlow context. We propose the FEBR (Flow lEvel Bandwidth pRovisioning) algorithm, which reduces the switch scheduling problem to multiple instances of fair queueing problems, each employing a well studied fair queueing algorithm. FEBR can tightly emulate the ideal Generalized Processing Sharing model, and accurately guarantee the provisioned bandwidth. Further, we implement FEBR in the OpenFlow version 1.0 software switch. In conjunction with the capability of OpenFlow to flexibly define and manipulate flows, we thus provide a practical flow level bandwidth provisioning solution. Finally, we present extensive simulation and experiment data to validate the analytical results and evaluate our design.**

***Keywords*-OpenFlow; CICQ switches; bandwidth provisioning.**

I. Introduction

Bandwidth provisioning on switches offers bandwidth assurance for certain types of traffic [1], [2]. The objective is to emulate the ideal Generalized Processor Sharing (GPS) model [3], where each traffic flow has an independent logical transmission channel with its desired bandwidth. Based on the traffic unit, bandwidth provisioning can be at different granularity levels [1]. Port level bandwidth provisioning assures bandwidth for the traffic from an input to an output, while flow level bandwidth provisioning guarantees bandwidth for an individual flow, which may be a subset of the traffic from the input to the output. Bandwidth provisioning at the flow level is necessary, and is especially important for virtual network based experiment environments like GENI [4]. In such an environment, multiple virtual systems may reside in a single physical box, and their traffic departs from the same physical network adapter. Flow level bandwidth provisioning is able to isolate traffic of different virtual systems and ensure accurate experiment results.

Existing flow level bandwidth provisioning algorithms [1], [2], [5] suffer from a number of drawbacks. *First*, they have high hardware complexity and time complexity. Specifically, they require a crossbar with speedup of at least two, i.e. the crossbar having twice bandwidth as that of the input or output, and they may need large expensive on-chip memories for the crossbar. In addition, they run in a centralized mode with up to N iterations for an $N \times N$ switch. *Second*, they cannot achieve constant service guarantees. Constant service guarantees mean that for any flow, the difference between its service amount in an algorithm and in the ideal Generalized Processor Sharing (GPS) model [3] is bounded by constants, i.e. the equations in Theorem 1 of [6], and they are the key properties to assure worst-case fairness. The reason is that [7] the existing algorithms cannot emulate WF^2Q [6] (including its variants), the only fair queueing algorithm to achieve constant service guarantees. *Third*, the existing algorithms can only handle fixed length cells. When variable length packets arrive, they have to be first segmented into fixed length cells at inputs. The cells are then transmitted to outputs, where they are reassembled into original packets before sent to the output lines. This process is called segmentation and reassembly (SAR) [8], which may waste bandwidth due to padding bits [9].

In this paper, we study the flow level bandwidth provisioning problem in the OpenFlow [10] context. OpenFlow is an open standard to allow researchers to run experimental protocols in realistic networks, and is currently deployed in large-scale testbeds like GENI [11]. OpenFlow provides a rich set of options to define flows based on a combination of packet header fields, and use a flow table to allow users to flexibly control their traffic. Bandwidth provisioning has been recognized as an essential component of OpenFlow, to isolate traffic between different experiments or even different types of traffic within a single experiment [10]. However, the current OpenFlow implementation supports only token bucket based traffic shaping, which is necessary but not sufficient to provide tight performance guarantees [6].

We first propose the FEBR (Flow lEvel Bandwidth pRovisioning) algorithm for combined-input-crosspoint-queued (CICQ) switches, which are special crossbar switches with a small exclusive buffer at each crosspoint [12]. The crosspoint buffers decouple inputs and outputs, and greatly simplify the scheduling process. FEBR reduces the switch scheduling problem to multiple instances of fair queueing problems [13], each utilizing a well studied fair queueing algorithm. As a result, FEBR can tightly emulate the ideal GPS model and accurately guarantee the provisioned bandwidth.

In addition, we implement the FEBR algorithm in the OpenFlow version 1.0 software switch [14] as a practical flow level bandwidth provisioning solution. An OpenFlow network consists of a number of OpenFlow switches and one or more

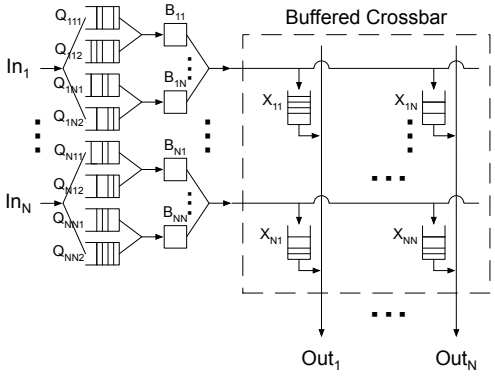


Fig. 1. Structure of CICQ Switches

controllers, and the controller communicates with the switches through the OpenFlow protocol. With the implementation of FEBR, the controller is able to arbitrarily define a flow, and inform the OpenFlow switches on the routing path to provision the desired bandwidth.

The rest of the paper is organized as follows. In Section II, we present our flow level bandwidth provisioning algorithm. In Section III, we describe the implementation of our algorithm in the OpenFlow software switch. In Section IV, we show simulation and experiment data to evaluate our design. Finally, in Section V, we conclude the paper.

II. Flow Level Bandwidth Provisioning for CICQ Switches

In this section, we present the FEBR algorithm for CICQ switches and analyze its performance.

A. Problem Formulation

The considered CICQ switch structure is shown in Figure 1. The switch has N inputs and N outputs, connected by a buffered crossbar without speedup. For easy representation, denote the i^{th} input as In_i and the j^{th} output as Out_j. Each input or output has bandwidth of R, and so does the crossbar. For flow level bandwidth provisioning, it is necessary to separate the traffic of different flows, i.e. storing incoming packets on a per flow basis. Denote the k^{th} flow from In_i to Out_j as F_{ijk}, and the queue at In_i to store its packets as Q_{ijk}. Besides the queue for each flow, In_i has a VOQ [15] buffer for each Out_j, denoted as B_{ij}, to store a packet before sending it to the crossbar. Each crosspoint of the crossbar has a small exclusive buffer. Denote the crosspoint buffer connecting In_i and Out_j as X_{ij}. Outputs have no buffers. Buffer management can be based on existing schemes, such as random early detection (RED) [16], and is out of the scope of this paper.

Our objective is to accurately provision bandwidth for each flow. Assume that a flow F_{ijk} has been provisioned with a certain amount of bandwidth R_{ijk}. In the ideal GPS model, F_{ijk} has a logical dedicated channel with exactly R_{ijk} bandwidth. Use $toO_{ijk}(0,t)$ and $\widehat{toO}_{ijk}(0,t)$ to represent the numbers of bits transmitted by F_{ijk} to the output during interval $[0,t]$ in our algorithm and GPS, respectively. Formally, the objective is to bound the absolute value of the difference $toO_{ijk}(0,t) - \widehat{toO}_{ijk}(0,t)$ by constants, independent of R_{ijk} and t.

B. Algorithm Description

The basic idea of FEBR is to reduce the switch scheduling problem to three stages of fair queueing, which we call flow scheduling, input scheduling, and output scheduling, respectively. *Flow scheduling* selects a packet from one of the flow queues Q_{ijk} from In_i to Out_j, and sends it to the corresponding VOQ buffer B_{ij}. *Input scheduling* selects a packet from one of the N VOQ buffers B_{ij} of In_i, and sends it to the corresponding crosspoint buffer X_{ij}. *Output scheduling* selects a packet from one of the N crosspoint buffers X_{ij} of Out_j, and sends it to the output line. The detailed description of each scheduling stage is as follows.

1) *Flow scheduling* utilizes the WF^{2}Q [6] fair queueing algorithm to multiplex different flows of the same input-output pair as a single logical flow, to simplify input scheduling. For easy description, denote the n^{th} packet of F_{ijk} as P^n_{ijk}. Flow scheduling calculates two time stamps for each packet p: virtual flow start time $\widehat{FS}(p)$ and finish time $\widehat{FF}(p)$. They are the departure time of the first bit and last bit of p in GPS, and are calculated as

$$\widehat{FS}(P^n_{ijk}) = \max(A(P^n_{ijk}), \widehat{FF}(P^{n-1}_{ijk})) \quad (1)$$

$$\widehat{FF}(P^n_{ijk}) = \widehat{FS}(P^n_{ijk}) + \frac{L(P^n_{ijk})}{R_{ijk}} \quad (2)$$

where $A(p)$ is the arrival time of p, and $L(p)$ is its packet length. Note that our objective is to accurately provision bandwidth, and we do not consider reallocating the leftover bandwidth of empty flows to backlogged flows. Thus, the virtual time in GPS progresses at the same pace as the real time, and the time stamp calculation is simpler than that in [6].

The *first step* of flow scheduling identifies eligible packets. A packet is eligible for flow scheduling if it has started transmission in GPS. Specifically, a packet p is eligible at time t if its virtual flow start time is less than or equal to t, i.e. $\widehat{FS}(p) \le t$. The *second step* selects among eligible packets the one p with the smallest virtual flow finish time. The selected packet will be sent to the corresponding VOQ buffer B_{ij}, to participate in input scheduling. If there are no eligible packets, flow scheduling will wait until the next earliest virtual flow start time. Additionally, we define two time stamps for p: actual flow start time $FS(p)$ and finish time $FF(p)$, to represent the actual departure time of its first bit and last bit from Q_{ijk} in flow scheduling. Flow scheduling multiplexes all flows from In_i to Out_j as a logical flow F_{ij}, which has bandwidth $R_{ij} = \sum_k R_{ijk}$. Thus, the last bit of p will leave Q_{ijk} at $FF(p) = FS(p) + L(p)/R_{ij}$.

Note that flow scheduling is only a logical operation to determine the sequence of packets to participate in input scheduling. There is no actual packet transmission for flow scheduling, because the packet is in the input buffer both before and after flow scheduling.

2) *Input scheduling* uses WF^{2}Q to multiplex the logical flows F_{ij} of the same input In_i to share the bandwidth to the crosspoint buffers. Input scheduling also calculates two time stamps for each packet p: virtual input start time $\widehat{IS}(p)$ and finish time $\widehat{IF}(p)$, which are equal to the actual flow start and finish time, respectively, i.e. $\widehat{IS}(p) = FS(p)$ and $\widehat{IF}(p) = FF(p)$. Similar as flow scheduling, the *first step* of input scheduling identifies eligible packets whose virtual input start time is no later than the current scheduling time. The *second step* finds among eligible packets the one with the smallest virtual input finish time. The selected packet is then sent from the VOQ buffer to the crosspoint buffer.

3) *Output scheduling* utilizes the WFQ [3] fair queueing algorithm to allow the crosspoint buffers of the same output to share the bandwidth to the output line. The reason to use WFQ instead of WF^{2}Q for output scheduling is that input scheduling has restricted admission of packets into the crosspoint buffers. Output scheduling uses only one time stamp for a packet p: virtual output finish time $\widehat{OF}(p)$, which can be calculated as $\widehat{OF}(p) = \widehat{IF}(p) + L_m/R + L_m/R_{ij}$, where L_m is the maximum packet length. Output scheduling simply retrieves the packet with the smallest virtual output finish time from the crosspoint buffers of an output and send it to the output line.

C. Performance Analysis

In this subsection, we present some analytical results on the performance of FEBR. Due to space limitations, we present only the main results and omit the proofs.

First, FEBR achieves accurately provisioned bandwidth, in the sense that the difference between the service amount of any flow in FEBR and GPS at any time is bounded by constants.

Theorem 1: At any time, the difference between the numbers of bits transmitted by a flow to the output in FEBR and GPS is greater than or equal to $-4L_m$ and less than or equal to L_m.

FEBR also achieves delay guarantees as stated by the following theorem.

Theorem 2: For any packet P^n_{ijk} , the difference between its departure time in FEBR and GPS is greater than or equal to $L(P^n_{ijk})(2/R - 1/R_{ijk})$ and less than or equal to $2L_m(1/R + 1/R_{ij})$

A nice feature of FEBR is that it has a size bound for the crosspoint buffers, which are expensive on-chip memories.

Theorem 3: In FEBR, the maximum number of bits buffered at any crosspoint buffer at any time is bounded by $3L_m$.

D. Implementation Advantages

FEBR is practical to implement with a number of advantages. First, FEBR can be implemented in a distributed manner, since there is no centralized scheduler, and different inputs or outputs do not need to exchange any information. The virtual output finish time of a packet can be calculated by the input and carried by the packet to the crosspoint buffer for output scheduling. Second, FEBR can directly process variable length packets without SAR. Compared with fixed length cell scheduling, variable length packet scheduling can achieve higher throughput and shorter latency [9] [17]. Finally, FEBR requires no speedup and has a small bounded crosspoint buffer size, reducing the hardware cost.

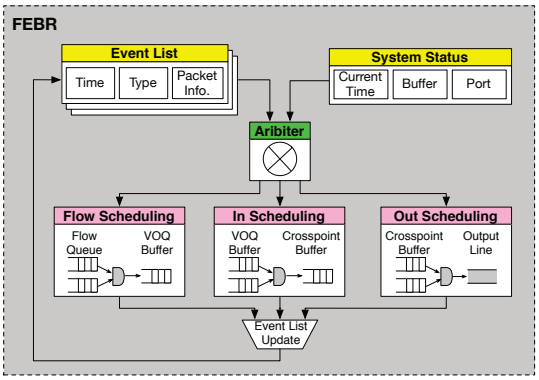


Fig. 2. Event-driven Scheduling Mechanism of FEBR enabled OpenFlow Software Switch

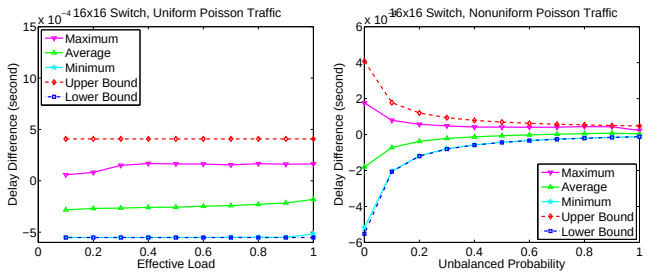
III. OpenFlow based Implementation

To evaluate our design in a realistic environment, we implement FEBR in the OpenFlow version 1.0 software switch [14], which converts a Linux PC with multiple NICs to an OpenFlow switch. In conjunction with the existing capability of OpenFlow to flexibly define and manipulate flows, we thus provide a practical flow level bandwidth provisioning solution. The original OpenFlow version 1.0 software switch is a user space program and acts as an OQ switch.

Because there is no concept of a crossbar in the original OpenFlow software switch, our first task is to create a virtual buffered crossbar to emulate the CICQ switch. We allocate space in the memory for the VOQ buffers B_{ij} and crosspoint buffers X_{ij}, and create the flow queues Q_{ijk} on demand, i.e. setting up a new flow queue when the controller creates a new entry in the flow table. We configure the bandwidth of the crossbar to be that of an input or output, and emulate the transmission delay from the VOQ buffer to the crosspoint buffer and from the crosspoint buffer to the output.

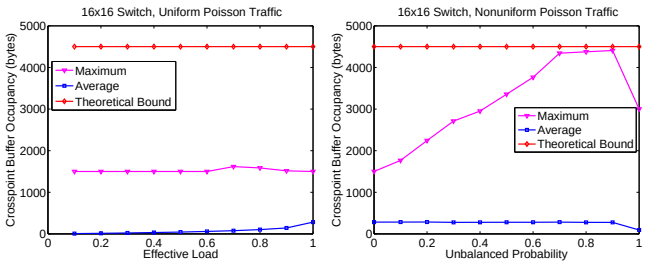
The next challenge is to maintain accurate system time. Since the original program needs no time stamps, the minimum time granularity is one millisecond. However, the operation of FEBR relies on time stamps, and requires maintaining accurate system time. Thus, the existing time granularity is not sufficiently fine, especially when the switch bandwidth is large. For example, if the switch bandwidth is 1 Gbps, the transmission time of a 1500 bytes packet is only 0.0015 millisecond. To address the challenge, we maintain accurate logical time within the virtual crossbar, to calculate correct time stamps for scheduling. Only the packet arrival time is based on the original system time, and all other operations of FEBR are based on the accurate logical time.

We extend the event driven mechanism of the original program to control the operation of the virtual crossbar, as illustrated in Figure 2. The original program uses an event



(a) (b)

Fig. 3. Delay Difference (a) Uniform Traffic (b) Nonuniform Traffic



(a) (b)

Fig. 4. Crosspoint Buffer Occupancy (a) Uniform Traffic (b) Nonuniform Traffic

driven mechanism, and monitors two types of events: packet arrival and time out. The program is normally blocked, and wakes up to process the assigned job when an event happens. We add all the possible types of events of the virtual crossbar to the event list, each with the necessary information, including the event time, event type, and associated packet. All the events are linked in an increasing order of the event time. When an event triggers, the program retrieves the first event in the event list and processes it. Note that processing an event may insert new events to the list. Because of the coarse granularity of the system time, multiple events may happen when the program wakes up, in which case the program will continue processing the event at the head of the event list until the time of the next event is in the future.

IV. Simulation and Experiment Results

We have implemented the FEBR algorithm in a Java simulator and the OpenFlow version 1.0 software switch. In this section, we present the numerical results from the simulations and experiments, to validate the analytical results and evaluate our design. Due to space limitations, we present only partial results.

A. Simulation Results

In the simulations, we consider a 16×16 CICQ switch without speedup. Each input and output has 1 Gbps bandwidth. There are two flows from In_i to Out_j with $R_{ij2} = 2R_{ij1}$. The packet length is uniformly distributed between 40 and 1500 bytes, and packets arrive based on a Markov modulated Poisson process with the same setting as in [17].

We use two traffic patterns. For traffic pattern one, or uniform traffic, we set $R_{ij} = R/N$, and change the effective load of the incoming traffic from 0.1 to 1 by step 0.1. For traffic pattern two, or nonuniform traffic, we fix the effective load to 1, and define R_{ij} by i, j and an unbalanced probability w as follows

$$R_{ij} = \begin{cases} R(w + \frac{1-w}{N}), & \text{if } i = j \\ R\frac{1-w}{N}, & \text{if } i \neq j \end{cases} \tag{3}$$

where w is increased from 0 to 1 by step 0.1.

Due to space limitations, we present only part of the results.

1) Delay Difference: Recall that Theorem 2 gives the upper bound and lower bound for the delay difference. Because the value of the lower bound depends on the length of each packet, it is not convenient to plot the figure. To eliminate the dependency, we calculate the lower bound for all packets as follows

$$L(P_{ijk}^n)(\frac{2}{R} - \frac{1}{R_{ijk}}) \geq \begin{cases} L_m(\frac{2}{R} - \frac{1}{R_{ijk}}), & \text{if } R_{ijk} \leq \frac{R}{2} \\ 0, & \text{if } R_{ijk} > \frac{R}{2} \end{cases} \tag{4}$$

Figure 3(a) shows the maximum, average, and minimum delay differences of one representative flow F_{111} under uniform traffic. As can be seen, the minimum delay difference is almost coincident with the lower bound, and the maximum delay difference is always less than the upper bound. The average delay difference is negative for all effective loads. Figure 3(b) plots the data under nonuniform traffic. We can see that the simulation data fall perfectly within the theoretical bounds. With the increase of the unbalanced probability, the maximum delay difference increases, and the minimum and average delay differences increase.

2) Crosspoint Buffer Occupancy: We now look at the crosspoint buffer occupancy data and compare them with Theorem 3. Figure 4(a) shows the maximum and average crosspoint occupancies under uniform traffic. As can be seen, the maximum crosspoint occupancy is less than the theoretical bound $3L_m$ for all the effective loads. In addition, the average crosspoint occupancy is always less than 400 bytes, much lower than the maximum value. Figure 4(b) presents the data under nonuniform traffic. We can see that the theoretical crosspoint buffer size bound is tight. Specifically, the maximum crosspoint occupancy increase constantly with the unbalanced probability, and drops to 3000 bytes when the unbalanced probability becomes one. The average crosspoint occupancy is close to 300 bytes and drop to around 100 bytes when unbalanced probability becomes one.

B. Experiment Results

We install the FEBR enabled OpenFlow software switch on Linux PCs for the following experiments. Each PC has an Intel Core 2 Duo 2.2 GHz processor, 2 GB RAM, and multiple 100 Mbps Ethernet NICs. The PC operating system is Ubuntu

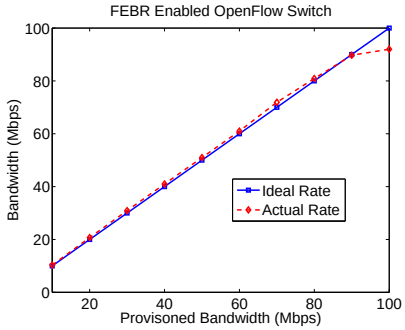


Fig. 5. Experiment with Single Flow and Single Switch

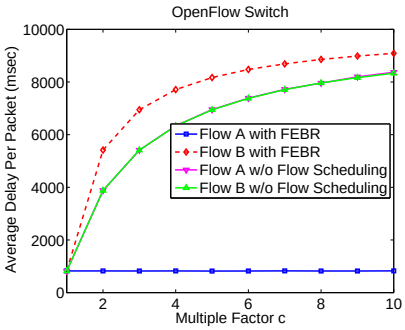


Fig. 6. Experiment with Multiple Flows and Single Switch

10.04LTS with Linux kernel version 2.6.33. NOX version 0.8 [18] is deployed as the OpenFlow controller.

1) Single Flow and Single Switch: In the first experiment, we compare the provisioned bandwidth of a flow with the measured bandwidth. We use a switch to connect two hosts, and set up an iperf [19] TCP flow between the two hosts. By TCP congestion control, the TCP flow can automatically probe the available bandwidth in the link. We adjust the provisioned bandwidth of the flow from 10 Mbps to 100 Mbps by step 10 Mbps. Note that because the NIC has maximum bandwidth of 100 Mbps, its ideal throughput is also 100 Mbps. As shown in Figure 5, when the provisioned bandwidth is less than 90 Mbps, the iperf measured bandwidth perfectly matches it. However, when the provisioned bandwidth becomes 100 Mbps, the measured bandwidth is only about 92.1 Mbps. The reasons might include the implementation overhead and the possibility that the NIC cannot reach its ideal throughput. As a comparison, the original OpenFlow software switch can achieve maximum bandwidth of about 94.5 Mbps.

2) Multiple Flows and Single Switch: In the second experiment, we compare FEBR with a port level bandwidth provisioning algorithm, i.e. without the flow scheduling phase. Similar as in the first experiment, a switch connects two hosts. There are now two iperf UDP flows between the two hosts, which we call Flow A and Flow B, and they share the same switch input and output. We provision each flow with 1 Mbps bandwidth. We fix the load of Flow A at 1 Mbps, and adjust the load the flow B from 1 Mbps to 10 Mbps by step 1 Mbps. As shown in Figure 6, with FEBR, the average delay of Flow A remains constant no matter what the load of Flow B is. The average delay of Flow B rises quickly, because it injects traffic at a higher rate than its provisioned bandwidth. On the contrary, with port level bandwidth provisioning, the average delay of both flows is coincident, and grows steadily with the load of Flow B. The results fully demonstrate that FEBR is effective in achieving traffic isolation among flows and providing flow level bandwidth provisioning.

V. Conclusions

In this paper, we have studied flow level bandwidth provisioning for CICQ switches in the OpenFlow context. We propose the FEBR algorithm and show that it can accurately emulate the ideal GPS model, and achieve constant service guarantees and tight delay guarantees. FEBR also has a number of implementation advantages, such as no speedup requirement, bounded crosspoint buffer sizes, distributed scheduling, and low time complexity. In addition, we implement FEBR in the OpenFlow software switch to provide a practical flow level bandwidth provisioning solution. Finally, we present extensive simulation and experiment data to validate the analytical results and evaluate our design.

References

[1] S. Chuang, S. Iyer, and N. McKeown, "Practical algorithms for performance guarantees in buffered crossbars," *IEEE INFOCOM 2005*, Miami, FL, Mar. 2005.

[2] S. Chuang, A. Goel, N. McKeown and B. Prabhkar, "Matching output queueing with a combined input output queued switch," *IEEE INFOCOM 1999*, New York, 1999.

[3] A. Parekh and R. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single node case," *IEEE/ACM Trans. Networking*, vol. 1, no. 3, pp. 344-357, Jun. 1993.

[4] Global Environment for Network Innovations, http://www.geni.net

[5] D. Pan and Y. Yang, "Providing flow based performance guarantees for buffered crossbar switches," *IEEE IPDPS 2008*, Miami, FL, Apr. 2008.

[6] J. Bennett and H. Zhang, "WF2Q: worst-case fair weighted fair queueing," *IEEE INFOCOM 1996*, San Francisco, CA, Mar. 1996.

[7] S. Iyer and N. McKeown, "Analysis of the parallel packet switch architecture," *IEEE/ACM Transactions on Networking*, vol. 11, no. 2, pp. 314-324, Apr. 2003.

[8] M. Katevenis and G. Passas, "Variable-size multipacket segments in buffered crossbar (CICQ) architectures," *IEEE ICC* , Korea, May 2005.

[9] J. Turner, "Strong performance guarantees for asynchronous crossbar schedulers," *IEEE/ACM Transactions on Networking*, vol. 17, no. 4, pp. 1017-1028, Aug. 2009.

[10] N. McKeown et al., "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69-74, Apr. 2008.

[11] GENI OpenFlow Backbone Deployment at Internet2, http://groups.geni.net/geni/wiki/OFI2

[12] I. Papaefstathiou, G. Kornaros, and N. ChrysosUsing, "Buffered crossbars for chip interconnection," *Great Lakes Symposium on VLSI*, Stresa-Lago Maggiore, Italy, Mar. 2007.

[13] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM 1989*, Austin, TX, Sep. 1989.

[14] OpenFlow 1.0 Release, http://www.openflowswitch.org/wk/index.php/OpenFlow_v1.0

[15] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input queued switch," *IEEE Trans. Commun.*, vol. 47, no. 8, pp. 1260-1267, Aug. 1999.

[16] S. Floyd and V. Jacobson "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking (TON)* vol. 1, no. 4, pp. 397-413, Aug. 1993

[17] D. Pan and Y. Yang, "Localized independent packet scheduling for buffered crossbar switches," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 260-274, Feb. 2009.

[18] NOX: An OpenFlow Controller, http://www.noxrepo.org

[19] IPerf: the TCP/UDP bandwidth measurement tool, http://sourceforge.net/projects/iperf/