OpenFlow-Based Flow-Level Bandwidth Provisioning for CICQ Switches

Hao Jin, *Student Member*, *IEEE*, Deng Pan, *Member*, *IEEE*,
Jason Liu, *Member*, *IEEE Computer Society*, and Niki Pissinou

Abstract—Flow-level bandwidth provisioning (FBP) achieves fine-grained bandwidth assurance for individual flows. It is especially important for virtualization-based computing environments such as data centers. However, 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 (CICQ) switches in the OpenFlow context. First, we propose the Flow-level Bandwidth Provisioning algorithm for CICQ switches, which reduces the switch scheduling problem to multiple instances of fair queuing problems, each utilizing a well-studied fair queuing algorithm. We theoretically prove that FBP can closely emulate the ideal Generalized Processing Sharing model, and accurately guarantee the provisioned bandwidth. Furthermore, we implement FBP in the OpenFlow software switch to obtain realistic performance data by a prototype. Leveraging the capability of OpenFlow to define and manipulate flows, we experimentally demonstrate a practical flow-level bandwidth provisioning solution. Finally, we conduct extensive simulations and experiments to evaluate the design. The simulation data verify the correctness of the analytical results, and show that FBP achieves tight performance guarantees. The experiment results demonstrate that our OpenFlow-based prototype can conveniently and accurately provision bandwidth at the flow level.

Index Terms—OpenFlow, CICQ switches, bandwidth provisioning

1 Introduction

Bandwidth provisioning on switches offers bandwidth assurance for designated traffic [1]. Based on the traffic unit, bandwidth provisioning can be at different granularity levels [2]. Port-level bandwidth provisioning assures bandwidth for the traffic from an input port to an output port, by which switches can support traffic isolation between VLANs. Since a switch port may belong to a single or multiple VLANs, bandwidth provisioning at the port level guarantees the bandwidth of each VLAN and makes one VLAN transparent to the other. On the other hand, flow-level bandwidth provisioning guarantees bandwidth for an individual flow, which may be a subset of the traffic from the input port to the output port. A flow may be the sequence of packets generated by a specific application or departing from an IP address, and in general can be flexibly defined by a combination of the 12 packet header fields [3].

Bandwidth provisioning at the flow level is necessary, as it differentiates traffic at sufficiently fine granularity [4]. It is particularly important for virtualization-based computing environments, such as data centers or the GENI-like [13] shared experimental infrastructure. In such an environment, multiple virtual machines (VMs) reside in a single physical server, and their traffic shares the same physical network adapter and is correspondingly fed into the same switch port. Flow-level bandwidth provisioning is able to isolate traffic of different VMs and make the shared underlying network infrastructure transparent to the VMs. The recently proposed Virtual Ethernet Port Aggregator (VEPA) protocol [5] off-loads all switching activities from hypervisor-based virtual switches to actual physical switches. As can be seen, VEPA requires flow-level bandwidth provisioning on switches to support traffic isolation between VMs.

Existing algorithms [1], [2], [6] achieve flow-level bandwidth provisioning by emulating Push-In-First-Out (PIFO) Output Queued (OQ) switches, but they 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 port or output port, 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, or in other words the scheduling time increases proportionally with the switch size. Second, they cannot achieve constant service guarantees. Constant service guarantees mean that for any flow, the difference between its service amount in a specific algorithm and in the ideal Generalized Processing Sharing (GPS) [7] model is bounded by constants, i.e., the equations in [8, Theorem 1]. The reason is that WF^{2}Q (including its variants) [8], the only known fair queuing algorithm to achieve constant service guarantees, does not use a PIFO queuing policy [9], and hence, the PIFO OQ switch emulation approach does not work. Third, although there have been switch designs [10] in the literature

- *H. Jin is with the Department of Electrical and Computer Engineering, Florida International University, Miami, FL 33199.*
E-mail: hjin001@fiu.edu.
- *D. Pan, J. Liu, and N. Pissinou are with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199.*
E-mail: {pand, liux, pissinou}@cis.fiu.edu.

Manuscript received 5 Dec. 2011; revised 18 June 2012; accepted 22 June 2012; published online 2 July 2012.
Recommended for acceptance by P. Bellavista.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-12-0935.
Digital Object Identifier no. 10.1109/TC.2012.167.

0018-9340/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

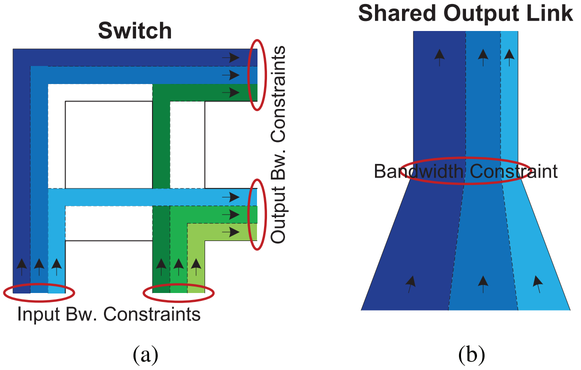


Fig. 1. GPS as Ideal Fairness Model (a) for switches, (b) for shared output links.

to directly handle variable length packets, the existing flow-level bandwidth provisioning algorithms can only handle fixed length cells. When variable length packets arrive, they have to be first segmented into fixed length cells at input ports. The cells are then transmitted to output ports, where they are reassembled into original packets before sent to the output links. This process is called segmentation and reassembly (SAR) [11], which wastes bandwidth due to padding bits [12].

In this paper, we study the flow-level bandwidth provisioning problem in the OpenFlow context [3]. Our objective is twofold: to design an efficient flow-level bandwidth provisioning algorithm with constant service guarantees, and to experimentally demonstrate a practical flow-level bandwidth provisioning solution based on the OpenFlow protocol. OpenFlow is an open protocol that gives access to the forwarding plane of switches and routers, so that users can control their traffic in the network. It has been deployed in large-scale testbeds like GENI [13], and considered in many recent data center designs [14], [15]. 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 of different users or different types [16].

In our OpenFlow-based bandwidth provisioning solution, there will be a central controller and a number of switches. On the one hand, the controller collects resource and request information from the switches, allocates bandwidth for flows, and updates the flow tables of switches to enforce the provisioned bandwidth. On the other hand, the switches receive flow definition and bandwidth allocation information from the controller, and run the proposed switch scheduling algorithm to guarantee the allocated bandwidth. The focus of this paper is for the switches to accurately guarantee the allocated bandwidth of each flow by emulating the ideal GPS model. In GPS, each flow has a virtual dedicated channel with the allocated bandwidth, as shown in Fig. 1a. Thus, there is no interference between different flows, and each flow always receives the exact amount of its allocated bandwidth. Our goal is to bound the difference between the service amount of any flow in our algorithm and in GPS by constants, or in other words to achieve constant service guarantees. A more detailed problem formulation will be presented in Section 3.1.

We first propose the Flow-level Bandwidth Provisioning (FBP) algorithm for Combined Input Crosspoint Queued (CICQ) switches. CICQ switches are special crossbar switches with an on-chip buffer at each crosspoint, which is made available by recent development in VLSI technology [17], [18], and thus they are also called buffered crossbar switches. We consider CICQ switches because the crosspoint buffers decouple input ports and output ports, and greatly simplify the scheduling process. Different from existing algorithms, FBP reduces the switch scheduling problem to multiple instances of fair queuing problems [19], each utilizing a well-studied fair queuing algorithm. As a result, FBP can closely emulate the ideal GPS model and accurately guarantee the provisioned bandwidth. Furthermore, we implement the FBP algorithm in the OpenFlow version 1.0 software switch [20] to obtain realistic performance data through a prototype. Besides FBP-enabled OpenFlow software switches, our prototype includes a NOX version 0.8 controller, which informs the switches on the flow path to provision the desired bandwidth. Leveraging the capability of the OpenFlow protocol to define and manipulate flows, we have thus demonstrated a practical solution to accurately provision bandwidth at the flow level.

Our main contributions in this paper are as follows: First, we theoretically analyze the performance of FBP, and prove that it achieves constant service guarantees and tight delay guarantees. Second, FBP is economical to implement with bounded crosspoint buffer sizes and no speedup requirement, and is fast with low time complexity and distributed scheduling. Third, we implement FBP in the OpenFlow software switch and integrate it with the NOX controller, to experimentally demonstrate a practical flow-level bandwidth provisioning solution. Fourth, we present extensive simulation and experiment data to show the effectiveness of our design.

The rest of the paper is organized as follows: In Section 2, we discuss the background and related work. In Section 3, we present the FBP algorithm. In Section 4, we theoretically analyze the performance of FBP. In Section 5, we describe the implementation of the OpenFlow-based prototype. In Section 6, we show simulation and experiment data to evaluate our design. Finally, in Section 7, we conclude the paper.

2 Background and Related Work

In this section, we provide an overview of existing bandwidth provisioning algorithms for switches, analyze their insufficiencies, and review the mechanisms currently available in OpenFlow.

2.1 Existing Bandwidth Provisioning Algorithms for Switches

The current main approach of bandwidth provisioning on switches is to emulate PIFO OQ switches. In a PIFO OQ switch, all packets are buffered at output ports, either on a per input port or per flow basis. Each output port runs a fair queuing algorithm [19] to emulate the ideal GPS model and provide guaranteed bandwidth for each output queue. OQ switches achieve the optimal performance but are not practical because they need speedup of N [21].

The following solutions provide port-level bandwidth provisioning. Magill et al. [22] show that a buffered crossbar switch with speedup of two satisfying nonnegative slackness insertion, lowest-time-to-live blocking, and lowest-time-to-live fabric scheduling can exactly emulate a PIFO OQ switch. Mhamdi and Hamdi [23] proposes the *Modified Current Arrival First—Lowest Time To Leave First* scheduling algorithm, for a one-cell buffered crossbar switch with speedup of two to emulate a PIFO OQ switch without time stamps. Chuang et al. [2] show that with speedup of two, a buffered crossbar switch can mimic a PIFO OQ switch with the restriction that the cells of an input-output pair depart in the same order as they arrive. He et al. [25] propose the rate-based *Smooth Multiplexing* algorithm for a CICQ switch with a two-cell buffered crossbar, and shows that the algorithm provides bandwidth and throughput guarantees. Turner [12] presents the *Packet Group by Virtual Output Queue (GBVOQ)* and *Packet Least Occupied Output First* scheduling algorithms for buffered crossbar switches, and shows that they can emulate PIFO OQ switches with speedup of two or more. Stoica and Zhang [24] propose the *Joined Preferred Matching* algorithm for CIOQ switches, and prove that the algorithm can emulate a general class of OQ service disciplines. Attiya et al. [26] propose frame-based schedulers for Combined Input Output Queued (CIOQ) switches handling variable length packets to mimic an ideal OQ switch with bounded delay, and demonstrates a tradeoff between the switch speedup and the relative queuing delay. Wu et al. [27] consider high-speed packet switches with optical fabrics, and propose scheduling algorithms to provide performance guaranteed switching. Kanizo et al. [28] introduce the Crosspoint Queued switch with large crosspoint buffers and no input queues, and propose scheduling algorithms for it to emulate an ideal OQ switch.

As mentioned earlier, existing flow-level bandwidth provisioning solutions need high hardware and time complexity. Chuang et al. [2] show that in order for a buffered crossbar switch with speedup of two to provide flow-level bandwidth provisioning, a separate crosspoint buffer must be available for each flow. Alternatively, the switch structure must first be modified with a more complicated buffering scheme (similar to that of OQ switches) and then a total of N^3 crosspoint buffers must be provided. Unfortunately, both schemes greatly increase the total number of crosspoint buffers and are not scalable. Another option is to increase the speedup of the crossbar to three, which will drop the maximum throughput of the switch by one-third. The additional speedup of one is used to eliminate the crosspoint blocking. Chuang et al. [1] propose several algorithms for CIOQ switches with speedup of two to emulate PIFO OQ switches. The *Critical Cell First* (CCF) algorithm needs N^2 iterations and global information. The *Delay Till Critical* (DTC) algorithm reduces the iteration number to N, but still needs global information. On the other hand, the *Group by Virtual Output Queue* (GBVOQ) algorithm does not need global information, but its iteration number is unbounded. Pan and Yang [6] present a scheme to achieve tradeoffs between those in [2] and [1]. It conducts distributed scheduling in the average case, but still needs speedup of two and N iterations in the worst case.

2.2 Insufficiencies of PIFO OQ Switch Emulation Approach

Although PIFO OQ switches cover a wide range of service disciplines such as WFQ [19] and DDR [29], there are a number of insufficiencies. Besides the high hardware and time complexity, inability to achieve constant service guarantees, and inefficiency to process variable length packets as discussed in Section 1, the bandwidth allocation policy of PIFO OQ switches, originated from fair queuing algorithms for shared output links, is not suitable for switches. Specifically, it fails to consider the bandwidth constraints at input ports, while flows may oversubscribe the input ports [30].

The objective of the emulation approach is to emulate a fair queuing algorithm at each output port. A fair queuing algorithm schedules packets from multiple flows of a shared output link to ensure fair bandwidth allocation, and it allocates bandwidth to the flows proportional to their requested bandwidth [7]. Specifically, assume that the available bandwidth of the shared output link is R, and ϕ_i and R_i are the requested bandwidth and allocated bandwidth of the ith flow, respectively. With proportional bandwidth allocation, we have $\forall i, \forall j, R_i/\phi_i = R_j/\phi_j$ and $\sum_i R_i \leq R$. However, simple proportional bandwidth allocation is not suitable for switches [31]. The reason is that while flows of a shared output link are constrained only by the output link bandwidth as in Fig. 1b, flows of a switch are subject to two bandwidth constraints: the available bandwidth at both the input port and output port of the flow, as shown in Fig. 1a. Naive bandwidth allocation at the output port may make the flows violate the bandwidth constraints at their input ports, and vice versa.

In the following, we use an example to illustrate the problem. Consider a 2×2 switch. For easy representation, denote the ith input port as In_i and the jth output port as Out_j. Assume that each input port or output port has available bandwidth of one unit. Use ϕ_{ij} and R_{ij} to represent the requested bandwidth and allocated bandwidth of In_i at Out_j, respectively, and ϕ_{ij} is initialized as in (1). Assume that each output port uses the proportional bandwidth allocation policy, i.e., the policy used by fair queuing algorithms for shared output links. First we look at only Out_1. Because $\phi_{11} = 0.9$ and $\phi_{21} = 0.6$, by the proportional policy we have $R_{11} = 0.6$ and $R_{21} = 0.4$. The same applies to Out_2. The allocated bandwidth R_{ij} is thus shown in (1). However, this allocation is not feasible, because the total bandwidth allocated at In_1 is $R_{11} + R_{12} = 0.6 + 0.6 = 1.2$, exceeding the available bandwidth of 1. For the same reason, if bandwidth allocation is conducted independently by each input port using the proportional policy, the allocation will not be feasible either:

$$\phi = \begin{bmatrix} 0.9 & 0.75 \\ 0.6 & 0.5 \end{bmatrix} \Rightarrow R = \begin{bmatrix} \mathbf{0.6} & \mathbf{0.6} \\ 0.4 & 0.4 \end{bmatrix}. \tag{1}$$

In addition, to improve utilization, fair queuing algorithms will reallocate the leftover bandwidth of empty flows using the proportional policy. In other words, when a flow temporarily becomes empty, the fair queuing algorithm will reallocate its bandwidth to the remaining backlogged flows in proportion to their requested bandwidth. However, this strategy does not apply to switches either, and we use the following example to explain. Consider the same 2×2 switch, and assume that the initial allocated bandwidth $\forall i \forall j, R_{ij} = 0.5$, as shown in (2). Now that In_1 temporarily has no traffic to Out_1, i.e., $R_{11} = 0.5$ changing to $R'_{11} = 0$. The proportional bandwidth allocation policy would allocate the leftover bandwidth of R_{11} to R_{21}, because now only In_2 has traffic to Out_1. However, this is not possible, because it will oversubscribe In_2 by 0.5. As a matter of fact, the leftover bandwidth of R_{11} cannot be reallocated at all in this case:

$$R = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix} \Rightarrow R' = \begin{bmatrix} 0 & 0.5 \\ \mathbf{1} & \mathbf{0.5} \end{bmatrix}. \tag{2}$$

From the above discussions, we can see that bandwidth allocation of the PIFO OQ switch emulation approach is not practical. In other words, the PIFO OQ switch emulation approach works only under predetermined feasible bandwidth allocation. As a result, we adopt a two-step approach, similar to that in [25], by separately addressing the bandwidth allocation and switch scheduling issues. In the first step, the only task of the bandwidth allocation algorithms is to assign a certain feasible amount of bandwidth for each flow. In our design, since the OpenFlow controller connects to every switch in an OpenFlow network, it has all the bandwidth and flow information. The controller can thus make optimal decisions for flow routing and bandwidth allocation. In the second step, given the allocated bandwidth of each flow, the task of the switch scheduling algorithms is to guarantee that each flow receives the exact amount of bandwidth. In our design, the OpenFlow switches will receive bandwidth allocation information from the controller, and run the FBP algorithm to achieve constant service guarantees.

In the following discussions, our focus will be the switch scheduling task, and we will not discuss the bandwidth allocation task in detail for the following reasons. First, as analyzed above, the proportional bandwidth allocation policy used by fair queuing algorithms is not applicable to switches. As a matter of fact, many existing PIFO OQ emulation-based solutions, such as [1], [2], [6], [12], [22], [23], and [24], have neither simulations nor experiments. They are certainly of theoretical interests, but it is difficult if not impossible to apply them in real networks. By comparison, we show a prototype in Section 5 to demonstrate that our solution is practical and readily available. Second, for proper bandwidth provisioning, the allocated bandwidth of a flow must consider all the switches and routers on the routing path. Allocating or adjusting bandwidth at a single hop is not likely to take effect. In this sense, the bandwidth allocation issue should be solved at the network level, and can be handled by the central controller in an OpenFlow network, or by protocols like RSVP [35]. Third, for bandwidth allocation on a single switch, the challenge has been addressed by solutions in the literature [25], [31], [33].

2.3 Bandwidth Provisioning in OpenFlow

Bandwidth provisioning is recognized as an essential component for OpenFlow [16]. The current OpenFlow implementation supports a Hierarchical Token Bucket (HTB) [32]-based framework called slicing, which is necessary but not sufficient to provide tight performance guarantees [8]. As stated in [16], slicing is a minimum but not complete QoS scheme. Slicing utilizes the HTB technique, which is a combination of token bucket traffic shaping and deficit round robin fair queuing [29]. HTB assures only minimal bandwidth, and cannot accurately guarantee the provisioned bandwidth. In addition, OpenFlow has a special controller called FlowVisor [34], which creates slices of network resources and provides traffic isolation between different slides.

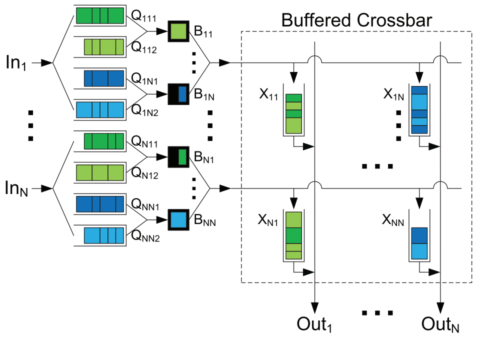


Fig. 2. Structure of CICQ switches.

3 Flow-Level Bandwidth Provisioning for CICQ Switches

In this section, we formulate the flow-level bandwidth provisioning problem, and present the FBP algorithm for CICQ switches.

3.1 Problem Formulation

The considered CICQ switch structure is shown in Fig. 2. The switch has N input ports and N output ports. Denote the ith input port as In_i and the jth output port as Out_j. The input ports and output ports are connected by a buffered crossbar without speedup. In other words, each input port or output port has bandwidth of R, and so does the crossbar. For flow-level bandwidth provisioning, it is necessary for input ports to separate the traffic of different flows, i.e., storing incoming packets on a per flow basis. Denote the kth flow from In_i to Out_j as F_{ijk}, and the queue at In_i to store its packets as Q_{ijk}. Besides a queue for each flow, In_i has a virtual output buffer for each Out_j, denoted as B_{ij}, to store the next packet departing from In_i to Out_j. Note that B_{ij} is not a physical buffer, but a pointer pointing to the head packet of one of the queues from In_i to Out_j. Each crosspoint of the crossbar has a small buffer. Denote the crosspoint buffer connecting In_i and Out_j as X_{ij}. There are no buffers at output ports.

Our objective is to accurately provision bandwidth for each flow by emulating the ideal GPS model. GPS views flows as fluids of continuous bits, and creates a virtual dedicated channel for each flow based on its allocated bandwidth, as shown in Fig. 1a. Because GPS is a fluid-based system, a flow can smoothly stream from the input port to the output port without buffering in the middle. We thus assume that packets in GPS will skip the virtual output buffers and crosspoint buffers. GPS is also the ideal packet scheduling model of fair queuing algorithms for shared output links, as shown in Fig. 1b.

Assume that a flow F_{ijk} has been allocated a certain amount of bandwidth R_{ijk}. Use $toO_{ijk}(0,t)$ and $\widehat{toO}_{ijk}(0,t)$ to represent the numbers of bits transmitted by F_{ijk} to the output port during interval $[0,t]$ in our algorithm and GPS, respectively. Formally, the objective is to bound the difference $|toO_{ijk}(0,t) - \widehat{toO}_{ijk}(0,t)|$ by constants, independent of R_{ijk} and t. Note that for feasible bandwidth allocation, no input or output should have oversubscription, i.e., $\forall i, \sum_{j,k} R_{ijk} \leq R$, and $\forall j, \sum_{i,k} R_{ijk} \leq R$. The feasibility requirement is only for bandwidth allocation. Temporary overload is allowed for any input port and output port, with overloading packets being temporarily stored in input buffers.

3.2 Algorithm Description

The basic idea of the FBP algorithm is to reduce the switch scheduling problem to three stages of fair queuing, 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 virtual output buffer B_{ij}. Input scheduling selects a packet from one of the N virtual output 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 port. The detailed description of each scheduling stage is as follows:

3.2.1 Flow Scheduling

Flow scheduling utilizes the WF^{2}Q [8] fair queuing 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 nth 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 $\widehat{FS}(P^n_{ijk}) = \max(A(P^n_{ijk}), \widehat{FF}(P^{n-1}_{ijk}))$ and $\widehat{FF}(P^n_{ijk}) = \widehat{FS}(P^n_{ijk}) + L(P^n_{ijk})/R_{ijk}$, where $A(p)$ is the arrival time of p, and $L(p)$ is the packet length.

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., $\widehat{FS}(p) \leq t$. The second step selects among eligible packets the one p with the smallest virtual flow finish time, i.e., $\forall p', \widehat{FS}(p') \leq t \rightarrow \widehat{FF}(p') \geq \widehat{FF}(p)$. The selected packet will be sent to the corresponding virtual output 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.

3.2.2 Input Scheduling

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 virtual output buffer to the crosspoint buffer. Additionally, we define the actual input start time $IS(p)$ and finish time $IF(p)$ to represent the time that the first bit and last bit of p leave B_{ij} in input scheduling, respectively. We have $IF(p) = IS(p) + L(p)/R$, since the bandwidth of the crossbar is R.

3.2.3 Output Scheduling

Output scheduling utilizes the WFQ [7] fair queuing algorithm to allow the crosspoint buffers of the same output to share the bandwidth to the output link. We can use WFQ instead of WF^{2}Q for output scheduling because 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 link. Additionally, define the actual output start time $OS(p)$ and finish time $OF(p)$ to represent the actual departure time of the first bit and last bit of p from X_{ij}. Since the bandwidth of the crossbar is R, we have $OF(p) = OS(p) + L(p)/R$.

4 Performance Analysis

We now analyze the performance of FBP, and will show that it achieves constant service guarantees, tight delay guarantees, and bounded crosspoint buffer sizes. Since the three scheduling stages of FBP use the well-studied WF^{2}Q [8] and WFQ [7] fair queuing algorithms, our analysis will leverage the existing results for them. Both WF^{2}Q and WFQ schedule packets of multiple flows to emulate the ideal GPS model, and share some features in common. As indicated by [8, Theorem 1] and [7, Theorem 1], there is an important property between the virtual departure time $\widehat{\mathcal{F}}(p)$ of a packet p in the virtual dedicated channel and the actual departure time $\mathcal{F}(p)$ in the physical multiplexed channel with bandwidth $\mathcal{R}$: $\mathcal{F}(p) \leq \widehat{\mathcal{F}}(p) + L_m/\mathcal{R}$.

Recall that flow scheduling uses WF^{2}Q to multiplex all the flows from In_i to Out_j, which share bandwidth of R_{ij}, as a logical flow. By the above property, we have $FF(p) \leq \widehat{FF}(p) + L_m/R_{ij}$. Input scheduling uses WF^{2}Q to multiplex the logical flows from In_i to different Out_j as an aggregate flow. For input scheduling, we can view the virtual input finish time $\widehat{IF}(p) (= FF(p))$ as the departure time of p in the virtual dedicated channel. Since the physical multiplexed channel for input scheduling, i.e., the channel from the input buffer to the crossbar, has bandwidth of R, we can obtain $IF(p) \leq \widehat{IF}(p) + L_m/R$.

Output scheduling uses WFQ to multiplex flows from different crosspoint buffers, and we show below that $\widehat{OF}(p)$ is the departure time of p in the virtual dedicated channel with bandwidth R_{ij} for packets from X_{ij} to Out_j. For easy representation, denote the nth packet from In_i to Out_j as P_{ij}^n, and define $\widehat{OS}(P_{ij}^n) = \widehat{OF}(P_{ij}^n) - L(P_{ij}^n)/R_{ij}$.

Lemma 1. *By $\widehat{OS}(P_{ij}^n)$, P_{ij}^{n-1} has left X_{ij} and P_{ij}^n has arrived at X_{ij} in the virtual dedicated channel.*

Proof. First, it is easy to see that $\widehat{OS}(p) \geq \widehat{IF}(p) + L_m/R \geq IF(p)$, which means that p has arrived at the crosspoint buffer by $\widehat{OS}(p)$, and thus can start transmission in the virtual dedicated channel. Second, by the definition

$$\begin{aligned}\widehat{OS}(P_{ij}^n) &= \widehat{IF}(P_{ij}^n) + \frac{L_m}{R} + \frac{L_m}{R_{ij}} - \frac{L(P_{ij}^n)}{R_{ij}} \\ &\geq \widehat{IF}(P_{ij}^{n-1}) + \frac{L_m}{R} + \frac{L_m}{R_{ij}} = \widehat{OF}(P_{ij}^{n-1}),\end{aligned} \tag{3}$$

we know that P_{ij}^{n-1} has left X_{ij} in the virtual dedicated channel by $\widehat{OS}(P_{ij}^n)$, and thus P_{ij}^n can start transmission without conflict. □

According to Lemma 1, we can safely view $\widehat{OS}(p)$ as the departure time of the first bit of p in the virtual dedicated channel, and thus $\widehat{OF}(p)$ is the departure time of the last bit of p in the virtual dedicated channel. Therefore, we have

$$OF(p) \leq \widehat{OF}(p) + \frac{L_m}{R}. \tag{4}$$

4.1 Service Guarantees

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

Define $toO_{ijk}(t_1, t_2)$, $toX_{ijk}(t_1, t_2)$, and $toB_{ijk}(t_1, t_2)$ to denote the numbers of bits transmitted by F_{ijk} during interval $[t_1, t_2]$ to Out_j, X_{ij}, and B_{ij} in FBP, respectively. Correspondingly, use $\widehat{toB}_{ijk}(t_1, t_2)$ to represent the number of bits transmitted by F_{ijk} to B_{ij} during $[t_1, t_2]$ in GPS. With the virtual dedicated channel of F_{ijk}, $\widehat{toB}_{ijk}(t_1, t_2)$ is also the number of bits sent by F_{ijk} to Out_j during $[t_1, t_2]$ in GPS.

Lemma 2. *When a packet P_{ijk}^n starts transmission to the output port in FBP, the number of bits transmitted to the output port by its flow F_{ijk} in FBP is greater than or equal to that in GPS minus $4L_m$, i.e., $toO_{ijk}(0, OS(P_{ijk}^n)) \geq \widehat{toB}_{ijk}(0, OS(P_{ijk}^n)) - 4L_m$.*

Proof. By the definition of $OS(P_{ijk}^n), P_{ijk}^{n-1}$ has finished output scheduling at $OS(P_{ijk}^n)$ in FBP, i.e.,

$$toO_{ijk}(0, OS(P_{ijk}^n)) = \sum_{a=1}^{n-1} L(P_{ijk}^a). \tag{5}$$

On the other hand,

$$\begin{aligned}&\widehat{toB}_{ijk}(0, OS(P_{ijk}^n)) \\ &= \widehat{toB}_{ijk}\left(0, OF(P_{ijk}^n) - \frac{L(P_{ijk}^n)}{R}\right) \\ &\leq \widehat{toB}_{ijk}\left(0, \widehat{OF}(P_{ijk}^n) + \frac{L_m}{R} - \frac{L(P_{ijk}^n)}{R}\right) \\ &= \widehat{toB}_{ijk}\left(0, \widehat{IF}(P_{ijk}^n) + \frac{2L_m}{R} + \frac{L_m}{R_{ij}} - \frac{L(P_{ijk}^n)}{R}\right) \\ &\leq \widehat{toB}_{ijk}\left(0, \widehat{FF}(P_{ijk}^n) + \frac{2L_m}{R} + \frac{2L_m}{R_{ij}} - \frac{L(P_{ijk}^n)}{R}\right) \\ &\leq \widehat{toB}_{ijk}(0, \widehat{FF}(P_{ijk}^n)) + R_{ijk}\left(\frac{2L_m}{R} + \frac{2L_m}{R_{ij}} - \frac{L(P_{ijk}^n)}{R}\right) \\ &= \sum_{a=1}^{n} L(P_{ijk}^a) + R_{ijk}\left(\frac{2L_m}{R} + \frac{2L_m}{R_{ij}} - \frac{L(P_{ijk}^n)}{R}\right).\end{aligned} \tag{6}$$

By (5) and (6), we have

$$\begin{aligned}&\widehat{toB}_{ijk}(0, OS(P_{ijk}^n)) - toO_{ijk}(0, OS(P_{ijk}^n)) \\ &\leq L(P_{ijk}^n) + 2L_m\frac{R_{ijk}}{R} + 2L_m\frac{R_{ijk}}{R_{ij}} - L(P_{ijk}^n)\frac{R_{ijk}}{R} \\ &= L(P_{ijk}^n)\left(1 - \frac{R_{ijk}}{R}\right) + 2L_m\frac{R_{ijk}}{R} + 2L_m\frac{R_{ijk}}{R_{ij}} \\ &\leq L_m\left(1 - \frac{R_{ijk}}{R}\right) + 2L_m\frac{R_{ijk}}{R} + 2L_m\frac{R_{ijk}}{R_{ij}} \\ &\leq L_m\left(1 + \frac{R_{ijk}}{R}\right) + 2L_m\frac{R_{ijk}}{R_{ij}} \\ &\leq 4L_m.\end{aligned}$$

□

The following theorem shows that FBP achieves constant service guarantees.

Theorem 1. *At any time, the difference between the numbers of bits transmitted by a flow to the output port in FBP and GPS is greater than or equal to $-4L_m$ and less than or equal to L_m, i.e., $-4L_m \leq toO_{ijk}(0, t) - \widehat{toB}_{ijk}(0, t) \leq L_m$.*

Proof. Without loss of generality, assume that $t \in [OF(P_{ijk}^n), OF(P_{ijk}^{n+1}))$. First, we prove $toO_{ijk}(0, t) - \widehat{toO}_{ijk}(0, t) \geq -4L_m$. If $t \in [OF(P_{ijk}^n), OS(P_{ijk}^{n+1}))$, by noting $toO_{ijk}(t, OS(P_{ijk}^{n+1})) = 0$, we have

$$\begin{aligned}&toO_{ijk}(0, t) - \widehat{toB}_{ijk}(0, t) \\ &\quad = \left(toO_{ijk}(0, OS(P_{ijk}^{n+1})) - toO_{ijk}(t, OS(P_{ijk}^{n+1}))\right) \\ &\qquad - \left(\widehat{toB}_{ijk}(0, OS(P_{ijk}^{n+1})) - \widehat{toB}_{ijk}(t, OS(P_{ijk}^{n+1}))\right) \\ &\quad = \left(toO_{ijk}(0, OS(P_{ij}^{n+1})) - \widehat{toB}_{ijk}(0, OS(P_{ijk}^{n+1}))\right) \\ &\qquad + \widehat{toB}_{ijk}(t, OS(P_{ij}^{n+1})) \\ &\quad \geq -4L_m + \widehat{toB}_{ijk}(t, OS(P_{ijk}^{n+1})) \\ &\quad \geq -4L_m.\end{aligned} \tag{8}$$

Otherwise, if $t \in [OS(P_{ij}^{n+1}), OF(P_{ij}^{n+1}))$, by noting $toO_{ijk}(OS(P_{ijk}^{n+1}), t) = (t - OS(P_{ijk}^{n+1}))R$, we have

$$\begin{aligned} & toO_{ijk}(0,t) - \widehat{toB}_{ijk}(0,t) \\ &\quad = \big(toO_{ijk}\big(0, OS\big(P_{ijk}^{n+1}\big)\big) + toO_{ijk}\big(OS\big(P_{ijk}^{n+1}\big), t\big)\big) \\ &\quad\quad - \big(\widehat{toB}_{ijk}\big(0, OS\big(P_{ijk}^{n+1}\big)\big) + \widehat{toB}_{ijk}\big(OS\big(P_{ijk}^{n+1}\big), t\big)\big) \\ &\quad \geq -4L_m + \big(t - OS\big(P_{ijk}^{n+1}\big)\big)R - \widehat{toB}_{ijk}\big(OS\big(P_{ijk}^{n+1}\big), t\big) \\ &\quad \geq -4L_m + \big(t - OS\big(P_{ijk}^{n+1}\big)\big)(R - R_{ijk}) \\ &\quad \geq -4L_m. \end{aligned} \tag{9}$$

Next, we prove $toO_{ijk}(0,t) - \widehat{toB}_{ijk}(0,t) \leq L_m$. Since flow scheduling uses WF^{2}Q, by [8, Theorem 1], we have $toB_{ijk}(0,t) \leq \widehat{toB}_{ijk}(0,t) + L_m$ and thus $toO_{ijk}(0,t) \leq toB_{ijk}(0,t) \leq \widehat{toB}_{ijk}(0,t) + L_m$. □

4.2 Delay Guarantees

FBP also achieves delay guarantees as stated by the following theorem. Note that $OF(p)$ and $\widehat{FF}(p)$ are the departure time of a packet p in FBP and GPS, respectively.

Theorem 2. *For any packet P_{ijk}^n, the difference between its departure time in FBP and GPS is greater than or equal to $L(P_{ijk}^n)(2/R - 1/R_{ijk})$ and less than or equal to $2L_m(1/R + 1/R_{ij})$, i.e., $L(P_{ijk}^n)(2/R - 1/R_{ijk}) \leq OF(P_{ijk}^n) - \widehat{FF}(P_{ijk}^n) \leq 2L_m(1/R + 1/R_{ij})$.*

Proof. First, we prove $OF(P_{ijk}^n) - \widehat{FF}(P_{ijk}^n) \geq L(P_{ijk}^n)(2/R - 1/R_{ijk})$. By the flow scheduling and input scheduling policies, we have $IS(p) \geq FS(p) \geq \widehat{FS}(p)$, or

$$IF\big(P_{ijk}^n\big) - \frac{L\big(P_{ijk}^n\big)}{R} \geq \widehat{FF}\big(P_{ijk}^n\big) - \frac{L\big(P_{ijk}^n\big)}{R_{ijk}}.$$

By the output scheduling policy, we know $OS(p) \geq IF(p)$, or

$$OF\big(P_{ijk}^n\big) - \frac{\big(P_{ijk}^n\big)}{R} \geq IF\big(P_{ijk}^n\big).$$

Combining the two equations, we have proved $OF(P_{ijk}^n) - \widehat{FF}(P_{ijk}^n) \geq L(P_{ijk}^n)(2/R - 1/R_{ijk})$.

Next, we prove $OF(P_{ijk}^n) - \widehat{FF}(P_{ijk}^n) \leq 2L_m(1/R + 1/R_{ij})$. By (4), we know

$$\begin{aligned} OF\big(P_{ijk}^n\big) &\leq \widehat{OF}\big(P_{ijk}^n\big) + \frac{L_m}{R} = \widehat{IF}\big(P_{ijk}^n\big) + \frac{L_m}{R} + \frac{L_m}{R_{ij}} + \frac{L_m}{R} \\ &\leq \widehat{FF}\big(P_{ijk}^n\big) + \frac{2L_m}{R} + \frac{2L_m}{R_{ij}}. \end{aligned}$$

□

4.3 Crosspoint Buffers

A nice feature of FBP is that it has a size bound for the crosspoint buffers, which are expensive on-chip memories. Define $toX_{ij}(t_1, t_2)$ and $toO_{ij}(t_1, t_2)$ to be the numbers of bits transmitted by F_{ij} during interval $[t_1, t_2]$ to X_{ij} and Out_j (i.e., out of X_{ij}) in FBP, respectively.

Lemma 3. *When a packet P_{ij}^n starts transmission to the output in FBP, the number of buffered bits at its crosspoint buffer X_{ij} is bounded by $3L_m$, i.e., $toX_{ij}(0, OS(P_{ij}^n)) - toO_{ij}(0, OS(P_{ij}^n)) \leq 3L_m$.*

Proof. By the definition of $OS(P_{ij}^n)$, P_{ij}^{x-1} has finished output scheduling at $OS(P_{ij}^n)$, i.e.,

$$toO_{ij}\big(0, OS\big(P_{ij}^n\big)\big) = \sum_{a=1}^{n-1} L\big(P_{ij}^a\big). \tag{11}$$

On the other hand,

$$\begin{aligned} & toX_{ij}\big(0, OS\big(P_{ij}^n\big)\big) \\ &\quad = toX_{ij}\left(0, OF\big(P_{ij}^n\big) - \frac{L\big(P_{ij}^n\big)}{R}\right) \\ &\quad \leq toX_{ij}\left(0, \widehat{OF}\big(P_{ij}^n\big) + \frac{L_m}{R} - \frac{L\big(P_{ij}^n\big)}{R}\right) \\ &\quad = toX_{ij}\left(0, \widehat{IF}\big(P_{ij}^n\big) + \frac{2L_m}{R} + \frac{L_m}{R_{ij}} - \frac{L\big(P_{ij}^n\big)}{R}\right). \end{aligned} \tag{12}$$

Define $\widehat{toX}_{ij}(0,t)$ to represent the number of bits sent by the logical flow F_{ij} in the virtual dedicated channel with bandwidth R_{ij} during interval $[0,t]$. Recall that input scheduling uses the WF^{2}Q scheduling algorithm. Thus, by [8, Theorem 1], we know $toX_{ij}(0,t) \leq \widehat{toX}_{ij}(0,t) + L_m(1 - R_{ij}/R)$, and

$$\begin{aligned} & toX_{ij}\big(0, OS\big(P_{ij}^n\big)\big) \\ &\quad \leq \widehat{toX}_{ij}\left(0, \widehat{IF}\big(P_{ij}^n\big) + \frac{2L_m}{R} + \frac{L_m}{R_{ij}} - \frac{L\big(P_{ij}^n\big)}{R}\right) \\ &\quad\quad + L_m\left(1 - \frac{R_{ij}}{R}\right) \\ &\quad \leq \widehat{toX}_{ij}\big(0, \widehat{IF}\big(P_{ij}^n\big)\big) + R_{ij}\left(\frac{2L_m}{R} + \frac{L_m}{R_{ij}} - \frac{L\big(P_{ij}^n\big)}{R}\right) \\ &\quad\quad + L_m\left(1 - \frac{R_{ij}}{R}\right) \\ &\quad = \sum_{a=1}^{n} L\big(P_{ij}^a\big) + R_{ij}\left(\frac{2L_m}{R} + \frac{L_m}{R_{ij}} - \frac{L\big(P_{ij}^n\big)}{R}\right) + L_m\left(1 - \frac{R_{ij}}{R}\right). \end{aligned} \tag{13}$$

By (11) and (13), we can obtain

$$\begin{aligned} & toX_{ij}\big(0, OS\big(P_{ij}^n\big)\big) - toO_{ij}\big(0, OS\big(P_{ij}^n\big)\big) \\ &\quad \leq L\big(P_{ij}^n\big) + 2L_m\frac{R_{ij}}{R} + L_m - L\big(P_{ij}^n\big)\frac{R_{ij}}{R} + L_m\left(1 - \frac{R_{ij}}{R}\right) \\ &\quad \leq L\big(P_{ij}^n\big)\left(1 - \frac{R_{ij}}{R}\right) + L_m\frac{R_{ij}}{R} + 2L_m \\ &\quad \leq L_m\left(1 - \frac{R_{ij}}{R}\right) + L_m\frac{R_{ij}}{R} + 2L_m \\ &\quad \leq 3L_m. \end{aligned}$$

□

The following theorem gives the bound of the crosspoint buffer size.

Theorem 3. *In FBP, the maximum number of bits buffered at any crosspoint buffer at any time is bounded by $3L_m$, i.e., $toX_{ij}(0,t) - toO_{ij}(0,t) \leq 3L_m$.*

TABLE 1
Comparison with Existing Algorithms

	More speedup, more buffers [1]	CCF, DTC, GBVOQ [2]	FBP
Service guarantees	$O(MN)$	$O(MN)$	$O(1)$
SAR for var. len. packets	Yes	Yes	No
Crossbar speedup	3, 2	2	1
# of crosspoint buffers	$O(N^2)$, $O(N^3)$	0	$O(N^2)$
Time complexity	$O(\log M + \log N)$	$O(\log M + N \log N)$, $O(\log M + \log N)$, unbounded	$O(\log M + \log N)$
Distributed scheduling	Yes	No, No, Yes	Yes

Proof. Without loss of generality, assume $t \in [OF(P_{ij}^{n}), OF(P_{ij}^{n+1}))$. If $t \in [OF(P_{ij}^{n}), OS(P_{ij}^{n+1}))$, we have

$$\begin{aligned}
&toX_{ij}(0,t) - toO_{ij}(0,t)\\
&\quad = toX_{ij}\big(0, OS\big(P_{ij}^{n+1}\big)\big) - toO_{ij}\big(0, OS\big(P_{ij}^{n+1}\big)\big)\\
&\qquad - toX_{ij}\big(t, OS\big(P_{ij}^{n+1}\big)\big) + toO_{ij}\big(t, OS\big(P_{ij}^{n+1}\big)\big)\\
&\quad \le 3L_m - toX_{ij}\big(t, OS\big(P_{ij}^{n+1}\big)\big)\\
&\quad \le 3L_m.
\end{aligned} \tag{15}$$

Otherwise, if $t \in [OS(P_{ij}^{n+1}), OF(P_{ij}^{n+1}))$

$$\begin{aligned}
&toX_{ij}(0,t) - toO_{ij}(0,t)\\
&\quad = toX_{ij}\big(0, OS\big(P_{ij}^{n+1}\big)\big) - toO_{ij}\big(0, OS\big(P_{ij}^{n+1}\big)\big)\\
&\qquad + toX_{ij}\big(OS\big(P_{ij}^{n+1}\big), t\big) - toO_{ij}\big(OS\big(P_{ij}^{n+1}\big), t\big)\\
&\quad \le 3L_m + toX_{ij}\big(OS\big(P_{ij}^{n+1}\big), t\big) - toO_{ij}\big(OS\big(P_{ij}^{n+1}\big), t\big)\\
&\quad \le 3L_m + toX_{ij}\big(OS\big(P_{ij}^{n+1}\big), t\big) - \big(t - OS\big(P_{ij}^{n+1}\big)\big)R\\
&\quad \le 3L_m.
\end{aligned}$$

□

4.4 Complexity Analysis

As can be seen, to transfer an incoming packet to the output link, flow scheduling, input scheduling, and output scheduling each is conducted once. The time complexity of both WF^{2}Q and WFQ has been shown to be $O(\log M)$ [19], [36] to schedule M flows. Assuming that an input-output pair has at most M flows, then the time complexity of flow scheduling is $O(\log M)$. The time complexity of input scheduling and output scheduling is the same $O(\log N)$, because each of the two scheduling stages handles N flows. Regarding space complexity, a packet needs two time stamps, for the virtual start time and finish time of a scheduling stage.

4.5 Implementation Advantages

FBP is practical to implement with a number of advantages. First, FBP can be implemented in a distributed manner, because there is no centralized scheduler, and different input ports or output ports need no information exchange. The virtual output finish time of a packet can be calculated based on its virtual input finish time by the input port and carried by the packet to the crosspoint buffer for output scheduling. Second, FBP can directly process variable length packets without SAR. Because of distributed scheduling, there is no synchronized operation between different input ports and output ports, and thus each can independently process packets of variable length one by one. Note that packets in most real networks are of variable length. Compared with fixed length cell scheduling, variable length packet scheduling can achieve higher throughput and shorter latency [12], [21]. Finally, FBP requires no speedup and has a small bounded crosspoint buffer size of $3L_m$, reducing the hardware cost.

4.6 Comparison with Existing Solutions

We summarize the comparison of FBP and existing flow-level bandwidth provisioning algorithms in [1] and [2] in Table 1. First of all, we can notice that only FBP achieves $O(1)$ service guarantees, and avoids SAR for variable length packets. Since the other algorithms emulate PIFO OQ switches that run fair queuing algorithms at output ports, their performance guarantees are proportional to the number of flows at the output port, i.e., $O(MN)$. Also, they can only schedule fixed length cells. Next, comparing FBP with the algorithms in [1], we can see that FBP needs less speedup and fewer crosspoint buffers. Finally, comparing FBP with the algorithms in [2], we can see that FBP achieves better time complexity and enables distributed scheduling. The tradeoff is that FBP uses the CICQ switch structure with N^2 crosspoint buffers.

5 OpenFlow-Based Implementation

As stated in the introduction, Section 1, our second objective is to build an experimental prototype based on FBP to demonstrate a practical flow-level bandwidth provisioning solution. The prototype includes two components: OpenFlow switches running the FBP algorithm, and a NOX [37] OpenFlow controller with a self-developed bandwidth provisioning component. On the one hand, we implement FBP in the OpenFlow version 1.0 software switch [20], which converts a Linux PC with multiple NICs to an OpenFlow switch. Implementing the FBP algorithm will enable the software switch to accurately guarantee the provisioned bandwidth at the flow level. On the other hand, we develop a NOX component as the control console for bandwidth provisioning, where the network administrator can define a flow and specify its allocated bandwidth. Leveraging the flow manipulation capability of the OpenFlow protocol, our prototype can flexibly define flows, allocate bandwidth, and ensure the allocated bandwidth. In the following, we describe the implementation detail. The realistic performance data obtained from the prototype will be presented in Section 6.

5.1 FBP-Enabled OpenFlow Software Switches

We implement FBP in the OpenFlow version 1.0 software switch, which is a user space program. In the earlier versions of the OpenFlow software switch, the *data path* that manages the flow table was implemented as a kernel module. Starting from version 1.0, the entire program is implemented in the user space. The advantages of such a user space implementation include flexible development environment and good portability, but the tradeoff is performance degradation caused by frequent context switches. Note that the main objective of the software switch is to provide a reference OpenFlow design for test and demonstration purposes, but not for use in production networks. Therefore, the software switch considers more about convenience of deployment and less about performance. In our case, the user space software switch allows us to develop the prototype faster and more economically than hardware switches.

The original software switch acts as a shared-memory OQ switch. When a packet arrives at the input NIC, the program copies the packet from the input NIC buffer to the main memory. It then searches the flow table for a matching flow for the packet. If there is a matching flow, the program will obtain the output NIC from the table entry, and immediately transfer the packet from the main memory to the output NIC buffer, from where the packet will be sent to the output link. Otherwise, if there is no matching flow, which means that the packet is the first one of a new flow, the program will forward the packet to the controller, and the controller will create a new entry in the flow table.

As can be seen, there is no concept of a crossbar in the original OpenFlow software switch, and thus 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 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 the same as that of the NIC and emulate the transmission delay from the VOQ buffer to the crosspoint buffer and from the crosspoint buffer to the output port. In the FBP-enabled OpenFlow software switch, after a packet arrives at the input NIC, it is immediately retrieved to the flow queue in the memory using the *netdev_recv* function. The packet is then transmitted through the virtual crossbar, and finally delivered to the output NIC using the *netdev_send* function. *netdev_recv* and *netdev_send* are existing functions of the *netdev* module that manages the NICs.

The next challenge is to maintain accurate time stamps. For most Linux systems, the minimum time resolution is 1 μs [38]. Further, to avoid excessive overhead by signal handling, the minimum time resolution provided by the *timeval* module of the original software switch is 1 ms. However, the effectiveness of FBP relies on accurate time stamps, and the existing time resolution is not sufficiently fine, especially for high-speed switches. For example, assume that the minimum time resolution is 1 μs, and a software switch equipped with Gigabit NICs has 1-Gbps bandwidth. For simplicity, also assume that F_{ij1} is the only flow of In_i and Out_j, and thus $R_{ij} = 1$ Gbps. If a packet P_{ij1}^n has length $L(P_{ij1}^n)$ of 400 bits, and its actual flow start time $FS(P_{ij1}^n)$ is 5 μs, then its actual flow finish time will be $FF(P_{ij1}^n) = FS(P_{ij1}^n) + L(P_{ij1}^n)/R_{ij} = 5 + 0.4 = 5.4\ \mu$s. However, since the minimum time resolution is 1 μs, there is no way to differentiate 5 and 5.4 μs, and we have to round the latter to the former, which means the departure of P_{ij1}^n from its flow queue takes no time. More importantly, the error caused by the coarse time resolution will accumulate over time. To address the challenge, we maintain accurate logical time within the virtual crossbar, so as to calculate correct time stamps for scheduling. Only the packet arrival time is based on the original system time, and all other operations of FBP are based on the accurate logical time. Specifically, when a packet p is retrieved from the input NIC buffer, we call the existing *time_msec* function in the *timeval* module to obtain the packet arrival system time $A(p)$, which is an integer with *ms* time resolution. We then convert the integer system time value to the logical time as a double-precision floating-point number, and represent all the subsequent time stamps used by FBP as double-precision floating-point numbers. When the packet is sent to the output NIC, we obtain the logical time for the actual output finish time $OF(p)$, and use it to deduct the logical time $A(p)$ to derive the delay as a double-precision floating-point number. In this way, all the scheduling decisions of FBP are based on the more accurate double-precision logical time.

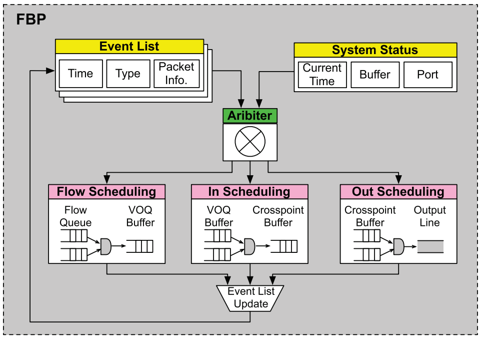


Fig. 3. Event-driven scheduling of FBP-enabled OpenFlow Software Switch.

Finally, we extend the event driven mechanism of the original software switch to control the operation of the virtual crossbar, as illustrated in Fig. 3. The original program uses an event 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 a timer 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 resolution 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.

5.2 Bandwidth Provisioning NOX Component

NOX is an open-source OpenFlow controller written in C++ and Python. The C++ code provides fundamental low-level APIs that are compliant with the OpenFlow protocol. The Python code implements the high-level control functionality, and interacts with the underlying C++ APIs. NOX enables customization with new functionality by adding new components written in Python.

We have developed a NOX component as the control console for the new bandwidth provisioning functionality. It accepts flow definition and bandwidth allocation as inputs, and sends OpenFlow commands as outputs to switches to set up flow table entries. For example, the network administrator can use the new NOX component to define the traffic from IP address 130.94.11.22 to 131.94.33.44 as a flow, and assign it 10-Mbps bandwidth. In our implementation, we use a configuration file to store all the flow definition and bandwidth allocation. The NOX component periodically checks the configuration file, and communicates the specified information to the OpenFlow switches. Each line of the configuration file contains 13 entries. The first 12 entries are the packet header fields to define a flow [39], and the last entry gives the allocated bandwidth of this flow. We add a timer for the NOX component to read the configuration file every five seconds, and use an array to store all the flow definition and bandwidth allocation information, with each array item corresponding to a defined flow.

Every time when the NOX component reads the configuration file, it compares the information read from the file with that already in the array. In the first case, if it detects a new flow defined in the configuration file, it adds the information to the array. When the first packet of the new flow arrives at a switch, the packet will be forwarded to the NOX controller. Our NOX component has a *packet_in_callback* function, which will be triggered by such a packet arrival event. By checking the packet header fields, the component recognizes that the packet belongs to the new defined flow, and uses the standard Layer 2 self-learning process to find a path for the flow. Next, the component adds a new entry in the flow table of each switch on the path, along with the provisioned bandwidth, by sending a flow table modification message of type *OFPFC_ADD*.

To send the provisioned bandwidth information from the controller to the switch, we need to modify the OpenFlow message format, the message sending function of the controller, and the message receiving function of the switch. First, we modify the flow modification message structure *ofp_flow_mod* in the *openflow.h* header file by adding a field named *bw* of type *uint32_t*. *openflow.h* defines the OpenFlow protocol format, and is shared by the controller and switch. To allow backward compatibility, for a regular flow without provisioned bandwidth, its *bw* field can be set to 0. Second, for the controller, we enhance the Python function *send_flow_command* in the *core.py* module and the C++ function *Pycontext::send_flow_command* in the *pycontext.cc* module, to add the bandwidth information to the message sent to the switch. Third, for the switch, when it receives the *OFPFC_ADD* message, it adds a new flow table entry with the provisioned bandwidth. In addition, to store the bandwidth information in the flow table, we modify the structure *sw_flow*, which stores all the information of a flow and is located in *switch-flow.h* header file, by adding a new filed named *bw* of type *uint32_t*. Future packets of the flow will match the newly added flow table entry, and will be transmitted using the provisioned bandwidth.

In the second case, if the component detects that an already defined flow was removed from the configuration file, it sends a flow table modification message of type *OFPFC_DELETE* to the switches to delete the corresponding flow table entry. Future packets of this flow will be processed by default without reserved bandwidth.

In the third case, if the component detects that the allocated bandwidth of a defined flow changed, it first updates the bandwidth in the array. Although the OpenFlow protocol defines a flow table modification message of type *OFPFC_MODIFY*, it can only modify the associated actions. Alternatively, our component sends a flow table modification message of type *OFPFC_DELETE* to delete the existing flow table entry of each switch. When the next packet of this flow arrives at a switch, the switch will treat it as if it was the first packet of a new flow and send it to the controller. The controller will then set up a new flow table entry for each switch, but with the updated bandwidth. Future packets of the flow will then be transmitted with changed bandwidth.

5.3 Scalability of OpenFlow-Based Implementation

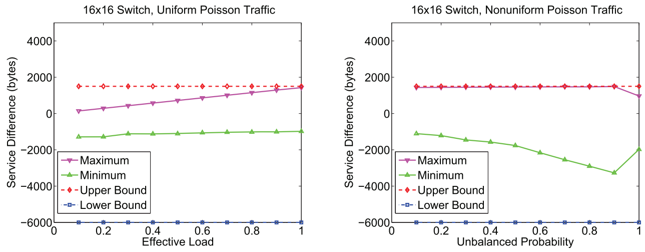
A good bandwidth provisioning solution needs to be scalable to support large numbers of flows and high traffic rates. We analyze the scalability of the OpenFlow-based implementation from the aspects of the switches and controller, respectively. For the switches, we have shown that our scheduling algorithms have low logarithmic time complexity, and thus can scale to high traffic rates. Further, it enhances scalability for switches of different roles to define flows at different granularity levels. Edges switches have only a small number of connected hosts, and thus can define a flow as the traffic generated by a single VM or application for flexible control. On the contrary, core switches handle enormous traffic, and the flows already shaped by edge switches can be combined as an aggregate flow to reduce management overhead. For the controller, it has been shown that an OpenFlow controller can handle all the flows of an enterprise network with tens of thousands of hosts [40]. In addition, OpenFlow has been considered in many recent data center designs [14], [15], and the experiments demonstrate the feasibility to use a central controller to manage large scale data centers. Finally, there are several recent proposals [41], [42] to scale the control of OpenFlow-like flow networks, and they can be utilized to enhance the scalability of the controller.

6 Simulation and Experiment Results

We have implemented the FBP algorithm in a simulator and the OpenFlow software switch. In this section, we present the numerical results from the simulations and experiments, to evaluate our design and validate the analytical results in Section 4.

6.1 Simulation Results

In the simulations, we consider a 16×16 CICQ switch without speedup. Each input port or output port has 1-Gbps bandwidth. There are two flows from In_i to Out_j with $R_{ij2} = 2R_{ij1}$, and thus the total number of flows is $16 \times 16 \times 2 = 512$. The packet length is uniformly distributed between 40 and 1,500 bytes, and packets arrive based on a Markov modulated Poisson process [21]. We use two traffic



(a) (b)

Fig. 4. Service difference. (a) uniform traffic (b) nonuniform 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:

$$R_{ij} = \begin{cases} R\left(w + \dfrac{1-w}{N}\right), & \text{if } i = j, \\ R\dfrac{1-w}{N}, & \text{if } i \neq j, \end{cases} \tag{17}$$

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

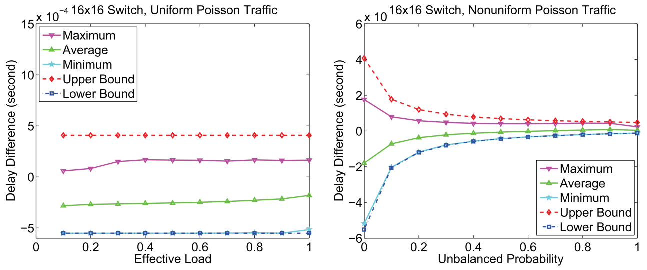
6.1.1 Service Guarantees

By Theorem 1, we know that the service difference of a flow in FBP and GPS at any time has a lower bound of $-4L_m$ and upper bound of L_m. We first look at the simulation data on service guarantees. Fig. 4a shows the maximum and minimum service differences among all the flows during the entire simulation run under uniform traffic. As can be seen, the maximum service difference increases with the traffic load, but does not exceed the theoretical upper bound. The minimum service difference is comparatively constant and always greater than the lower bound. The gap between the minimum service difference and the lower bound is caused by rounding the ratios of R_{ijk}/R_{ij} and R_{ijk}/R to integers in the proof of Lemma 2. In other words, the minimum service difference is determined by the bandwidth ratios but not the traffic load. Fig. 4b shows the simulation data under nonuniform traffic. We can see that the maximum service difference is almost coincident with the upper bound. Note that the maximum service difference drops when the unbalanced probability becomes one. The reason is that in this case, all packets of In_i go to Out_i. Thus, there is no switching necessary, and packet scheduling is only conducted between the two flows of the same input-output pair. Therefore, the maximum service difference is $L_m(R_{ij2}/R_{ij}) = 1{,}000$ bytes. On the other hand, the minimum service difference is always greater than the lower bound. It drops gradually when the unbalanced probability increases, and rises when the unbalanced probability becomes one, for the same reason as above. The low bound looks tighter under nonuniform traffic, because $\max_{i,j,k}\{R_{ijk}/R\}$ now has a greater value. The minimum service difference can keep getting closer to the lower bound by increasing the bandwidth ratios R_{ijk}/R_{ij} and R_{ijk}/R.

6.1.2 Delay Difference

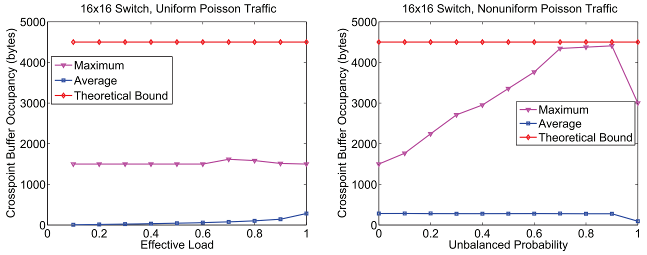
Recall that Theorem 2 gives the upper bound and lower bound for the delay difference of a flow in FBP and GPS. Because the lower bound value in the theorem depends on the lengths of individual packets, it is not convenient to plot the figure. To eliminate the dependency, we calculate the lower bound for all packets as follows:

$$L(P^n_{ijk})\left(\frac{2}{R} - \frac{1}{R_{ijk}}\right) \geq \begin{cases} L_m\left(\dfrac{2}{R} - \dfrac{1}{R_{ijk}}\right), & \text{if } R_{ijk} \leq \dfrac{R}{2}, \\ 0, & \text{if } R_{ijk} > \dfrac{R}{2}. \end{cases} \tag{18}$$



(a) (b)

Fig. 5. Delay difference. (a) uniform traffic, (b) nonuniform traffic.



(a) (b)

Fig. 6. Crosspoint buffer occupancy. (a) uniform traffic, (b) nonuniform traffic.

Fig. 5a 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. The maximum delay difference is always less than the upper bound, and has a small value. This shows that under uniform traffic, FBP can well emulate GPS and a packet will not depart too late after its departure time in GPS. Note that the average delay difference is less than zero for all effective loads, which means that most packets leave earlier in FBP than in GPS when the incoming traffic is uniformly distributed. Fig. 5b 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.

6.1.3 Crosspoint Buffer Occupancy

We now look at the crosspoint buffer occupancy data and compare them with Theorem 3. Fig. 6a 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. Fig. 6b presents the data under nonuniform traffic. We can see that the theoretical crosspoint buffer size bound is tight. Specifically, the maximum crosspoint occupancy increase

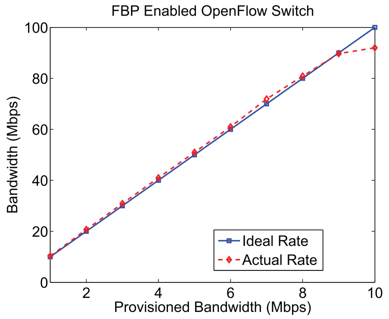


Fig. 7. Experiment with single flow and single switch.

constantly with the unbalanced probability, and drops to 3,000 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.

6.2 Experiment Results

We install the FBP 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 10.04LTS with Linux kernel version 2.6.33. NOX version 0.8 [37] is deployed as the OpenFlow controller.

6.2.1 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 [44] 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 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 Fig. 7, 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 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 without FBP can achieve maximum bandwidth of about 94.5 Mbps.

6.2.2 Multiple Flows and Single Switch

In the second experiment, we compare FBP 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 port and output port. We provision each flow with 1-Mbps bandwidth. We fix the load of Flow A at 1 Mbps, and adjust the load of Flow B from 1 to 10 Mbps by step 1 Mbps. As shown in Fig. 8, with FBP, 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 high rate than its provisioned bandwidth. On the contrary, with port level bandwidth provisioning, the average delay

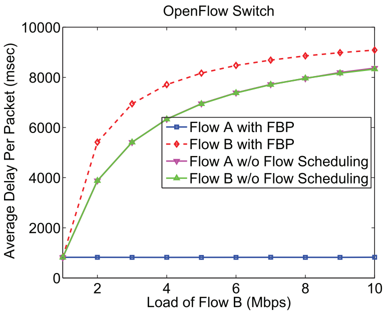


Fig. 8. Experiment with multiple flows and single switch.

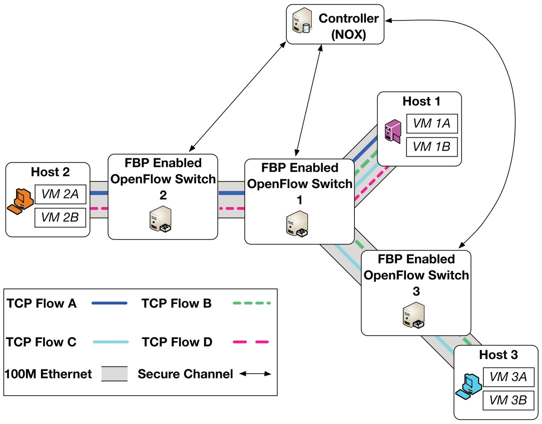


Fig. 9. Topology of experiment OpenFlow network.

of both flows grow steadily with the load of Flow B. The results fully demonstrate that FBP is effective in achieving traffic isolation among flows and providing flow-level bandwidth provisioning.

6.2.3 Multiple Flows and Multiple Switches

In the third experiment, we set up an OpenFlow network with one controller, three switches, and three hosts, with the topology shown in Fig. 9. Switch 1 connects Host 1, Switch 2, and Switch 3. Switch 2 connects Switch 1 and Host 2. Switch 3 connects Switch 1 and Host 3. Each host runs VirtualBox version 4.1.4 with two VMs. The VMs are configured with bridged networking [43] so that they will have public IP addresses. Denote the VMs on Host 1 as 1A and 1B, which emulate two TCP servers. Denote the VMs on Host 2 as 2A and 2B, and those on Host 3 as 3A and 3B, all emulating TCP clients. We set up four IPerf TCP flows: Flow A between VMs 1A and 2A, Flow B between VMs 1A and 3A, Flow C between VMs 1B and 3B, and Flow D between VMs 1B and 2B.

In the initial configuration, we set the provisioned bandwidth of Flows A, B, C, and D to be 15, 12, 8, and 6 Mbps, respectively. To measure the actual bandwidth of each flow, we install WireShark on Switch 1 to capture packets of all the four flows. Fig. 10 shows the continuous bandwidth measure of each flow by WireShark. Each pixel on the curve shows the average bandwidth of the flow during a 1-second interval. We can see that the measured bandwidth of each flows perfectly matches the provisioning amount, demonstrating that our solution is effective in a multiswitch and multiflow environment.

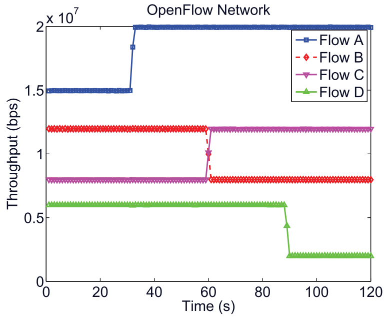


Fig. 10. Experiment with multiple flows and multiple switches.

Before the 30th second, we modify the configuration to increase the provisioned bandwidth of Flow A to 20 Mbps. When the NOX component reads the configuration, it detects the changed bandwidth allocation, and sends a command to the switches to realize this change. As can be seen from the figure, the measured bandwidth of Flow A quickly changes from 15 to 20 Mbps, and the measured bandwidth of the other flows remains the same. In a similar manner, before the 60th second, we modify the configuration to exchange the provisioned bandwidth amounts of Flows B and C, and before the 90th second, we reduce the provisioned bandwidth of Flow D to 2 Mbps. We can see that the prototype successfully handles all the bandwidth change requests, with the bandwidth of designated flows smoothly changing to the new values, and the bandwidth of the remaining flows keeping stable.

7 Conclusions

Flow-level bandwidth provisioning ensures allocated bandwidth for individual flows, and is especially important for virtualization-based computing environments such as data centers. However, existing solutions suffer from a number of drawbacks, including high hardware and time complexity, inability to achieve constant service guarantees, and inefficiency to process variable length packets. In this paper, we have studied flow-level bandwidth provisioning for CICQ switches in the OpenFlow context. First, we propose the FBP algorithm, which reduces the scheduling problem on CICQ switches to multiple stages of fair queuing, with each stage utilizing a well-studied fair queuing algorithm. We show by theoretical analysis that FBP can closely emulate the ideal GPS model, and achieve constant service guarantees and tight delay guarantees. FBP is economical to implement with bounded crosspoint buffer sizes and no speedup requirement, and is fast with low time complexity and distributed scheduling. In addition, we implement FBP in the OpenFlow software switch to build an experimental prototype. In conjunction with the existing capability of OpenFlow to flexibly define and manipulate flows, we have thus demonstrated a practical flow-level bandwidth provisioning solution. Finally, we conduct extensive simulations and experiments to evaluate our design. The simulation data successfully validate the analytical results, and the experiment results demonstrate that our prototype can accurately provision bandwidth at the flow level.

Acknowledgments

This work was partially supported by the US National Science Foundation under Grant CNS-1117016.

References

[1] S. Chuang, A. Goel, N. McKeown, and B. Prabhkar, “Matching Output Queueing with a Combined Input Output Queued Switch,” *Proc. IEEE INFOCOM,* Mar. 1999.
[2] S. Chuang, S. Iyer, and N. McKeown, “Practical Algorithms for Performance Guarantees in Buffered Crossbars,” *Proc. IEEE INFOCOM*, Mar. 2005.
[3] N. McKeown et al., “OpenFlow: Enabling Innovation in Campus Networks,” *ACM SIGCOMM Computer Comm. Rev.*, vol. 38, no. 2, pp. 69-74, Apr. 2008.
[4] N. Gude et al., “NOX: Towards an Operating System for Networks,” *ACM SIGCOMM Computer Comm. Rev.*, vol. 38, no. 3, pp. 105-110, July 2008.
[5] Edge Virtual Bridge Proposal, http://ieee802.org/1/files/public/docs2008/new-congdon-vepa-1108-v01.pdf, 2013.
[6] D. Pan and Y. Yang, “Providing Flow Based Performance Guarantees for Buffered Crossbar Switches,” *Proc. IEEE Int'l Symp. Parallel Distributed Processing (IPDPS)*, 2008.
[7] 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, June 1993.
[8] J. Bennett and H. Zhang, “WF^{2}Q: Worst-Case Fair Weighted Fair Queueing,” *Proc. IEEE INFOCOM*, Mar. 1996.
[9] S. Iyer and N. McKeown, “Analysis of the Parallel Packet Switch Architecture,” *IEEE/ACM Trans. Networking*, vol. 11, no. 2, pp. 314-324, Apr. 2003.
[10] D. Simos, I. Papaefstathiou, and M. Katevenis, “Building an FoC Using Large, Buffered Crossbar Cores,” *IEEE Design & Test of Computers*, vol. 25, no. 6, pp. 538-548, Nov. 2008.
[11] M. Katevenis and G. Passas, “Variable-Size Multipacket Segments in Buffered Crossbar (CICQ) Architectures,” *Proc. IEEE Int'l Conf. Comm. (ICC)*, May 2005.
[12] J. Turner, “Strong Performance Guarantees for Asynchronous Crossbar Schedulers,” *IEEE/ACM Trans. Networking*, vol. 17, no. 4, pp. 1017-1028, Aug. 2009.
[13] GENI OpenFlow Backbone Deployment at Internet2, http://groups.geni.net/geni/wiki/OFI2, 2013.
[14] R. Mysore et al., “Portland: A Scalable Fault-Tolerant Layer2 Data Center Network Fabric,” *Proc. ACM SIGCOMM Conf. Data Comm.*, Aug. 2009.
[15] M. Al-Fares et al., “Hedera: Dynamic Flow Scheduling for Data Center Networks,” *Proc. Seventh USENIX Conf. Networked Systems Design and Implementation (NSDI)*, Apr. 2010.
[16] OpenFlow Slicing, http://www.openflowswitch.org/wk/index.php/Slicing, 2013.
[17] G. Kornaros, “BCB: A Buffered Crossbar Switch Fabric Utilizing Shared Memory,” *Proc. Ninth EUROMICRO Conf. Digital System Design: Architectures, Methods Tools*, Aug. 2006.
[18] I. Papaefstathiou, G. Kornaros, and N. ChrysosUsing, “Buffered Crossbars for Chip Interconnection,” *Proc. Great Lakes Symp. VLSI*, Mar. 2007.
[19] A. Demers, S. Keshav, and S. Shenker, “Analysis and Simulation of a Fair Queueing Algorithm,” *Proc. ACM SIGCOMM*, 1989.
[20] OpenFlow 1.0 Release, http://www.openflowswitch.org/wk/index.php/OpenFlow_v1.0, 2013.
[21] D. Pan and Y. Yang, “Localized Independent Packet Scheduling for Buffered Crossbar Switches,” *IEEE Trans. Computers*, vol. 58, no. 2, pp. 260-274, Feb. 2009.
[22] B. Magill, C. Rohrs, and R. Stevenson, “Output-Queued Switch Emulation by Fabrics with Limited Memory,” *IEEE J. Selected Areas Comm.*, vol. 21, no. 4, pp. 606-615, May 2003.
[23] L. Mhamdi and M. Hamdi, “Output Queued Switch Emulation by a One-Cell-Internally Buffered Crossbar Switch,” *Proc. IEEE GLOBECOM*, Dec. 2003.
[24] I. Stoica and H. Zhang, “Exact Emulation of an Output Queueing Switch by a Combined Input Output Queueing Switch,” *Proc. IEEE/IFIP Sixth Int'l Workshop Quality of Service (IWQoS '98)*, May 1998.

[25] S. He, S. Sun, H. Guan, Q. Zheng, Y. Zhao, and W. Gao, "On Guaranteed Smooth Switching for Buffered Crossbar Switches," *IEEE/ACM Trans. Networking,* vol. 16, no. 3, pp. 718-731, June 2008.

[26] H. Attiya, D. Hay Member, and I. Keslassy, "Packet-Mode Emulation of Output-Queued Switches," *IEEE Trans. Computers,* vol. 59, no. 10, pp. 1378-1391, Oct. 2010.

[27] B. Wu, K. Yeung, M. Hamdi, and X. Li, "Minimizing Internal Speedup for Performance Guaranteed Switches with Optical Fabrics," *IEEE/ACM Trans. Networking,* vol. 17, no. 2, pp. 632-645, Apr. 2009.

[28] Y. Kanizo, D. Hay, and I. Keslassy, "The Crosspoint-Queued Switch," *Proc. IEEE INFOCOM,* Apr. 2009.

[29] M. Shreedhar and G. Varghese, "Efficient Fair Queuing Using Deficit Round Robin," *IEEE/ACM Trans. Networking,* vol. 4, no. 3, pp. 375-385, June 1996.

[30] X. Zhang, S. Mohanty, and L. Bhuyan, "Adaptive Max-Min Fair Scheduling in Buffered Crossbar Switches without Speedup," *Proc. IEEE INFOCOM,* May 2007.

[31] D. Pan and Y. Yang, "Max-Min Fair Bandwidth Allocation Algorithms for Packet Switches," *Proc. IEEE Int'l Symp. Parallel Distributed Processing (IPDPS),* Mar. 2007.

[32] Hierachical Token Bucket Theory, http://luxik.cdi.cz/devik/qos/htb/manual/theory.htm, 2013.

[33] M. Hosaagrahara and H. Sethu, "Max-Min Fairness in Input-Queued Switches," *IEEE Trans. Parallel and Distributed Systems,* vol. 19, no. 4, pp. 462-475, Apr. 2008.

[34] FlowVisor, http://www.openflowswitch.org/wk/index.php/FlowVisor, 2013.

[35] Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification, http://www.ietf.org/rfc/rfc2205.txt, 2013.

[36] P. Valente, "Exact GPS Simulation with Logarithmic Complexity, and Its Application to an Optimally Fair Scheduler," *IEEE/ACM Trans. Networking,* vol. 15, no. 6, pp. 1454-1466, Dec. 2007.

[37] NOX: An OpenFlow Controller, http://www.noxrepo.org, 2013.

[38] Man Page for gettimeofday, http://www.kernel.org/doc/man-pages/online/pages/man2/gettimeofday.2.html, 2013.

[39] OpenFlow Switch Specification Version 1.0.0, http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf, 2013.

[40] M. Casado et al., "Ethane: Taking Control of the Enterprise," *Proc. ACM SIGCOMM,* Aug. 2007.

[41] M. Yu, J. Rexford, M. Freedman, and J. Wang, "Scalable Flow-Based Networking with DIFANE," *Proc. ACM SIGCOMM,* Aug. 2010.

[42] A. Curtis et al., "DevoFlow: Scaling Flow Management for High-Performance Networks," *Proc. ACM SIGCOMM,* Aug. 2011.

[43] VirtualBox Virtual Networking, http://www.virtualbox.org/manual/ch06.html, 2013.

[44] IPerf: The TCP/UDP Bandwidth Measurement Tool, http://sourceforge.net/projects/iperf/, 2013.

Hao Jin received the BS degree in electrical engineering from Nanjing University, China, in 2006, and is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, Florida International University. His research interests include high-performance switch design and data center networking. He is a student member of the IEEE.

Deng Pan received the BS and MS degrees in computer science from Xi'an Jiaotong University, China, in 1999 and 2002, respectively, and the PhD degree in computer science from the State University of New York at Stony Brook, in 2007. He is currently an assistant professor in the School of Computing and Information Sciences, Florida International University. His research interests include high-performance switch architecture and high speed networking. He is a member of the IEEE.

Jason Liu received the BA degree from Beijing University of Technology in China in 1993, the MS degree from College of William and Mary in 2000, and the PhD degree from Dartmouth College in 2003. He is an associate professor at the School of Computing and Information Sciences, Florida International University. His research focuses on parallel simulation and high-performance modeling of computer systems and communication networks. He served as a general chair for MASCOTS 2010, SIMUTools 2011 and PADS 2012, and also as a program chair for PADS 2008 and SIMUTools 2010. He is an associate editor for *Simulation Transactions of the Society for Modeling and Simulation International*, and a Steering Committee member for PADS. He is a member of the IEEE Computer Society.

Niki Pissinou received the BS degree in industrial and systems engineering from The Ohio State University, the MSc degree in computer science from the University of California, Riverside, and the PhD degree from the University of Southern California. She is currently a professor in the School of Computing and Information Sciences, Florida International University. Her current research interests include high-speed networking, insider attacks detection and prevention in mobile ad hoc networks, and trust, privacy and data cleaning mechanisms in trajectory sensor networks.