Padded Frames: A Novel Algorithm for Stable Scheduling in Load-Balanced Switches

Juan José Jaramillo, *Student Member, IEEE*, Fabio Milan, *Student Member, IEEE*, and R. Srikant, *Fellow, IEEE*

***Abstract*—The load-balanced Birkhoff-von Neumann switching architecture consists of two stages: a load balancer and a deterministic input-queued crossbar switch. The advantages of this architecture are its simplicity and scalability, while its main drawback is the possible out-of-sequence reception of packets belonging to the same flow. Several solutions have been proposed to overcome this problem; among the most promising are the Uniform Frame Spreading (UFS) and the Full Ordered Frames First (FOFF) algorithms. In this paper, we present a new algorithm called Padded Frames (PF), which eliminates the packet reordering problem, achieves 100% throughput, and improves the delay performance of previously known algorithms.**

***Index Terms*—Birkhoff-von Neumann switch, load-balanced switch, scheduling.**

I. Introduction

THE continued explosive growth of the Internet fuels research in the problem of designing scalable, fast switching architectures. The change from output-queued switches to input-queued switches reduced the speedup of the switching fabric from N to 1, where N is the number of input ports. The solution to the Head-of-Line Blocking problem through the introduction of virtual output queues (VOQs) made it possible to achieve 100% throughput with input-queued switches, using queue-length based algorithms to resolve contention among cells [1], [2]. The Maximum Weight Matching Algorithm can find a solution to this problem but, unfortunately, has a complexity of $O(N^{2.5}\log N)$ [3], which makes it difficult to implement in practice. Several heuristics have been proposed to lower the complexity to $O(N\log N)$ with Maximal Matching [4] at the cost of increased speedup in the switch fabric, or $O(\log N)$ with randomized algorithms at the cost of increasing cell delay significantly [5], [6]. Another possible solution is to trade off a lower complexity with a greater delay by using a frame-based matching algorithm [7], [8], which runs only periodically once every few time slots.

An entirely different approach is the load-balanced Birkhoff-von Neumann algorithm (see Fig. 1), first introduced by Chang *et al.* in [9] and [10], which is implemented using a two-stage architecture. The first stage is a load balancer, which spreads the input traffic uniformly among the intermediate VOQs. The second stage is a deterministic input-queued crossbar switch. Both stages execute the same sequence of N configurations, such that every input and every intermediate queue is served deterministically $1/N$th of the time. It has been proved that this switch can provide 100% throughput under a broad class of traffic patterns. The main drawback is that, due to the existence of N parallel paths from each input i to each output j, it is possible for packets belonging to the same flow (i,j) to get out of order. To avoid having to reorder the packets at the output, several modifications to the basic load-balanced architecture have been proposed. See Section III for an overview of these algorithms.

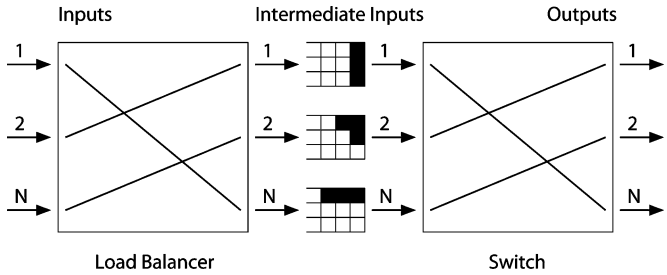

Fig. 1. Basic load-balanced architecture.

The contribution of this paper is threefold. First, we present a new approach to solve the reordering problem in a load-balanced switch. Our new algorithm is called Padded Frames (PF). Second, we compare the performance of PF with two of the most promising algorithms for this architecture, Uniform Frame Spreading (UFS) and Full Ordered Frames First (FOFF), which were first proposed by Keslassy *et al.* [11]. Third, we compare the performance of PF against algorithms for other architectures, specifically the ideal output-queued switch and iSLIP (which is designed for input-queued switches).

The rest of the paper is organized as follows. Section II describes the load-balanced switching architecture. Section III presents a brief overview of existing algorithms and their advantages and disadvantages. Section IV describes the Padded Frames algorithm. In Section IV-A we prove that our algorithm is stable under *any* admissible traffic pattern. In Section V we propose a modification to the original PF algorithm which enhances its performance, and in Section V-A we prove that it is stable. Section VI presents some simulations that confirm the analysis, and shows the performance improvement of Padded Frames with respect to existing algorithms. Section VII contains some comments about the implementation complexity of PF. Finally, Section VIII provides concluding remarks and presents some future research directions.

Manuscript received June 6, 2006; revised March 23, 2007. First published March 12, 2008; current version published October 15, 2008. Approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. Fumagalli. The research reported here was supported by a Fulbright fellowship to the first author and by the National Science Foundation under Grant CCF 06-34891. The second author's participation in the project occurred during a visit to the University of Illinois.

J. J. Jaramillo and R. Srikant are with the Department of Electrical and Computer Engineering, and the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801 USA (e-mail: jjjarami@uiuc.edu; rsrikant@uiuc.edu).

F. Milan is with the Dipartimento di Elettronica, Politecnico di Torino, 10129 Turin, Italy (e-mail: milan@mail.tlc.polito.it).

Digital Object Identifier 10.1109/TNET.2007.906654

II. Load-Balanced Switch Architecture

In this section, we present a brief overview of the load-balanced architecture to help the reader understand the PF algorithm to be presented later. A load-balanced switch is formed by two identical $N \times N$ switching stages, and a set of N^2 FIFO buffers at the input side of the second switching stage. We refer to the inputs at the second stage as *intermediate inputs*. At every intermediate input we have N buffers, called virtual output queues (VOQ), one per output. Each switching stage goes through the same predetermined cyclic shift configuration, so that every input is connected to each intermediate input (and every intermediate input is connected to each output) exactly $1/N$th of the time. A possible way to do so is to connect input (intermediate input) i, at time t, to intermediate input (output) $[(i+t-1) \bmod N]+1$, if we index inputs, intermediate inputs and outputs from 1 to N.

To understand how a load-balanced switch works, it may be helpful to follow the path of a packet arriving at input i and directed to output j. If it is assumed that only one packet arrives in one time slot at each input port, then no queues are needed at the inputs of the load-balanced switch. Indeed, immediately after its arrival, the packet is placed in some intermediate input k, depending on the time of its arrival, as described in the previous paragraph. More precisely, a packet directed to output j is stored in the jth VOQ of the kth intermediate input. Then, some time later, depending on the occupancy of the queue, the packet will be transferred through the second stage, from intermediate input k to its final destination j, where it will be transmitted on the output link.

In [9] it was proved that a load-balanced Birkhoff-von Neumann switch can provide 100% throughput for a wide class of arrival traffic scenarios, including Bernoulli i.i.d. traffic, on-off Markov bursty traffic, and Fractional Gaussian Noise traffic [12]. The fact that a completely deterministic switch can adapt to a stochastic arrival process may seem counterintuitive. However, this can be intuitively explained as follows. Consider a traditional single stage, input queued, $N \times N$ crossbar switch. If every input VOQ is served exactly one time slot every N, with a cyclic deterministic sequence as described above, then such a switch can achieve 100% throughput if the input traffic is uniform Bernoulli i.i.d. Note that nothing can be guaranteed if the traffic is not uniform. If we consider again the two-stage switch, it is now easy to see that, even if the input arrival process is not uniform, the input traffic seen by the second stage is uniform *enough* to be sustainable by the switch, because the packets have been spreaded uniformly over the intermediate input by the load balancer.

Since the load-balanced switch is completely deterministic, it does not require any centralized scheduler to choose which input to serve. Due to the absence of a scheduling algorithm, its complexity is not dependent on the number of inputs, i.e., it is $O(1)$. Hence, this architecture is suitable for the design of scalable, high performance switches.

However, the low complexity of the load-balanced switch comes at a cost. The main drawback of this architecture is that packets from a flow can get reordered within the switch. Indeed, due to the presence of N parallel paths for every input-output pair, two packets belonging to the same input-output flow will be generally stored in two different intermediate input queues.

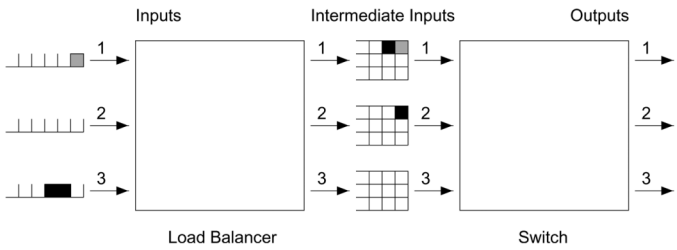

Fig. 2. Example of packet reordering in a load-balanced switch. The state of the intermediate input queues is presented at the end of time slot 2.

Note that the VOQ in which they are placed depends only on their destination, but not on their input port. Hence, different inputs may send packets to the same intermediate queue. It follows that there is no guarantee that two different intermediate VOQs associated to the same output have the same length.

The simple example of Fig. 2, adapted from [13], can be helpful to illustrate the reordering phenomenon. Let us consider a 3×3 load-balanced switch. Assume that the traffic pattern is as follows. At time 0 input 1 receives a packet directed to output 1 which is immediately sent to intermediate input 1. At times 1 and 2 input 3 receives two packets directed to output 1 that are sent to intermediate inputs 1 and 2 respectively. Since the packet that arrived *later* finds a shorter queue, it will be transferred *earlier* through the second stage, and the packet flow is reordered.

It would be natural to think about putting the packets in a resequencing buffer placed at the output of the switch, before sending them on the output link. However, consider the fact that the length of this resequencing buffer depends on the maximum amount of reordering suffered by the packets, which depends on the maximum difference between the lengths of the intermediate VOQs. Now, it can be easily seen that this length difference cannot be bounded. Imagine repeating the example over a long period of time. In this case the length of the first queue tends to diverge, and so does the amount of reordering. Since the amount of reordering in a simple load-balanced switch cannot be bounded, it is not feasible to simply solve the problem *a posteriori*. Hence, it is necessary to introduce some algorithm to solve the problem. The Section III presents a brief overview of solutions that have been proposed so far.

III. Existing Algorithms

To overcome the packet reordering problem in a load-balanced switch, without losing its simplicity and scalability, several solutions have been proposed. In this section we briefly review them.

A. First Come First Served

With the First Come First Served (FCFS) algorithm [10], packets are stored in the intermediate inputs according to both their arrival times and their flows. The intermediate inputs act as jitter control buffers, and ensure that the packets arrive in order at the output, so that no resequencing buffer is needed at the switch output. The main advantage of FCFS is that its computational complexity is independent of the switch size. However, it requires the intermediate buffers to have a speedup equal to N, so its implementation may be quite hard.

B. Earliest Deadline First

The Earliest Deadline First (EDF) algorithm [10], [14] assigns to every packet a target departure time, i.e., the time in which the packet should leave an ideal output-buffered switch. Then, packets are scheduled not according to their arrival time (as with FCFS), but according to their time stamp. Because EDF needs to pick up the packet with the minimum target departure time among all the stored packets, it is difficult to implement in a high performance switch.

C. Full Frames First

The Full Frames First (FFF) algorithm [15] is an attempt to simplify EDF by eliminating the resequencing buffer. To do so, no time stamp is calculated, but, to maintain order, packets are stored at the intermediate inputs according to their input-output pair. As a consequence, the intermediate buffer becomes much more complex, since 3-D queues are needed.

D. Application Flow-Based Routing

The idea of the Application Flow-Based Routing (AFBR) algorithm [13] is that reordering is harmful to network performance only if it occurs among packets belonging to the same application flow. So, AFBR hashes the header field of every packet (source and destination IP addresses and protocol identification) to get a value from 1 to N. Then, this hash value is used to determine to which intermediate input the packet will be sent. Since all packets belonging to the same application flow take the same path through the switch, reordering among them is avoided. The main drawback of AFBR is that it performs well only if the traffic pattern is made up of several, and small, applications flows. Otherwise, no throughput guarantee can be given.

E. Uniform Frame Spreading

The Uniform Frame Spreading (UFS) algorithm [13] prevents reordering by adopting a frame-based approach. At every input UFS needs N VOQs, one per each output, where packets are buffered. Then, an input is serviced only when it has at least N packets (i.e., a *frame*) waiting in the queue. Every packet of the frame will be placed in a different intermediate input, from 1 to N. The second stage works exactly the same as in a basic load-balanced switch.

It is easy to see that with UFS all the intermediate inputs have the same length. Thus, no reordering occurs in the switch, because every packet suffers the same queuing delay independently of the path it takes from its input to the output. Moreover, it has been proved in [13] that UFS achieves 100% throughput under *any* admissible traffic pattern.

The main disadvantage of the UFS algorithm is that packets may experience a long delay. Since an incomplete frame is not allowed to be scheduled, waiting for the frame to get full could lead to a significant delay, or even starvation of the less loaded flows traversing the switch.

F. Full Ordered Frames First

The Full Ordered Frames First (FOFF) algorithm [11], [13] allows a certain amount of reordering in order to reduce the delay suffered by packets. As in the case of UFS, FOFF requires the presence of N VOQs at every input, to store packets according to their output. FOFF will serve full frames whenever it is possible, but when no full frames are available, FOFF will serve the other queues in a round-robin order. Also in this case, the second stage works exactly the same as in a basic load-balanced switch.

Note that if there is always a full frame to schedule, FOFF is exactly the same as UFS, and there is no reordering. Reordering takes place only when an incomplete frame is served. However, it has been proved [13] that with FOFF the amount of reordering is always bounded. Hence, reordered packets can be ordered correctly by a *finite* buffer placed at the output of the switch.

As in UFS, under FOFF packets leave the switch in order. Moreover, FOFF guarantees 100% throughput for *any* admissible input traffic scenario. Finally, since flows can be served even if they do not have full frames in queue, the delay suffered by packets with FOFF, at lower loads, is less than the delay with UFS. In this paper we will study both UFS and FOFF using simulations and compare them to the PF algorithm that we propose here. See Section VI for the details.

IV. Padded Frames Algorithm

Padded Frames is a frame-based scheduling algorithm for load-balanced switching architectures. We use the term *frame* to refer to either a set of N packets, or to a set of N time slots, where N is the size of the switch, i.e., the number of input and output ports in the switch. Every input of the first stage is equipped with N VOQs. An arriving packet to input i directed to output j is placed in VOQ_{ij}. PF prevents packet reordering by assuring that all the intermediate input queues have the same length. In order to do so, every N time slots PF schedules for service with higher priority those input queues which have a full frame, choosing among them in round-robin order. If there is always a full frame to be transferred at each input port, PF behaves exactly as the UFS algorithm. When no full frames can be found at an input port, then PF chooses the largest nonempty queue from the VOQs at the input. As in the FOFF algorithm, serving the VOQs which do not have a complete frame, albeit at a lower priority, avoids long delays if the load is low, or starvation of smaller flows.

The distinctive feature of our algorithm is that, in order to deliver packets in sequence to the outputs, if it is necessary to transfer an incomplete frame of length $0 < F < N$; then PF will insert $N - F$ fake packets to obtain a *padded frame*. Sending such fake packets could eventually lead to an increase in the arrival rate to the intermediate inputs, and therefore produce instability. One of the contributions of this paper is to show that if the amount of padding is controlled, then this does not result in either instability or throughput loss.

To achieve this, we regulate the amount of padding present in the intermediate inputs with the use of a threshold T: we schedule a padded frame destined to output k only if the length of the intermediate input VOQs for output k is less than T. The idea is that the lengths of the queues are a form of congestion indication. Indeed, if the intermediate queues are long, then transferring fake packets could result in a throughput loss. On the other hand, if the intermediate queues are short, then the fake packets do not adversely impact switch stability. The intuitive reason for the stability of PF is that padding is necessary only at

light loads, so it does not result in a throughput reduction. Conversely, at heavy load, the probability of having a full frame is high; hence padding is not needed nor desired.

From an architectural point of view, since PF prevents packet reordering, it does not require an output buffer to resequence the packets.

As in the load-balanced Birkhoff-von Neumann switch, in PF each of the two switching stages goes through the same predetermined cyclic shift configuration. The chosen configuration in our algorithm will assure that every input will send packets starting from intermediate input 1 all the way through intermediate input N in an increasing order, and that every output will serve intermediate inputs in the same order.

Scheduling proceeds according to the following steps.

1) An arriving packet to input i directed to output j is placed in input VOQ_{ij}.
2) Each of the two switching stages of the load-balanced switch goes through the following predetermined cyclic shift configuration: at time t, input i is connected to intermediate input $[(i+t-1) \bmod N]+1$ and intermediate input $[(j+t-2) \bmod N]+1$ is connected to output j.
3) At every input, and every N time slots, the algorithm will search in round-robin order among the input VOQs that have at least N packets (one full frame).
4) If there exists no such input VOQ, the algorithm searches among the nonempty queues looking for the largest one. Without loss of generality, consider only input i. Assume that input VOQ_{ik} is the largest queue. Let L_k be the length of the kth VOQ of intermediate input 1, and let T be an integer parameter. Then perform the following test. If $L_k < T$, then schedule input VOQ_{ik} to be served. Else, do not serve input VOQ_{ik}.
5) When spreading packets from an input to the intermediate inputs, always start sending packets to intermediate input 1, and then continue serving intermediate inputs in an increasing order until intermediate input N is reached. When serving an input VOQ that was scheduled in step 4), if there is no packet to send, send a fake packet to the intermediate inputs. Otherwise, send the stored packets.
6) In the second stage, PF uses N VOQs per intermediate input (one per each output), and behaves exactly like the basic load-balanced switch: when intermediate input m is connected to output n, intermediate input VOQ_{mn} will be served.

To help clarify the algorithm, please refer to Appendix A for a pseudocode description of it.

Before concluding this section, we make a few comments on the PF algorithm. First, notice that if we set $T = 0$, then we are back to the UFS algorithm. Thus, PF is a generalization of UFS. Second, notice that when we are performing the test on the queue length to limit the number of padded frames in the intermediate input, we only have to look at the number of packets in intermediate input 1. Since packets are uniformly spread on all the N intermediate inputs, and either 0 or N (real or fake) packets are transferred, all the intermediate inputs have the same length. This information can be very useful in reducing the amount of information exchanged by the line cards. Finally, a comment on step 5). Suppose that a packet P directed to output j arrives at input i, exactly when the input i is serving VOQ_{ij}. Suppose that the length of the input VOQ_{ij} is strictly less than N, so that padding is necessary. Then, in this case, packet P can be used to preempt the fake packet which was scheduled to be transfered. Generally, this means that fake packets could be intertwined among real packets, thus improving the efficiency of the switch.

A. Stability

As is the case in AFBR, FFF, UFS, and FOFF, for simplicity we assume that all incoming packets are segmented into fixed-size cells, which we will simply call packets, and then reassembled at the output before leaving the switch. We also assume that there is at most one arrival per time slot at each input, and arrivals occur first in a time slot before departures.

We will prove that our algorithm achieves 100% throughput by comparing the load-balanced switch using the PF algorithm with the ideal output-queued switch.

The following lemma, proved in [16], will be useful in our proof.

Lemma 1: Consider a work-conserving server, and let $A(t)$ and $D(t)$ respectively denote its cumulative number of arrivals and services until time t. Assume that its service capacity is one packet per time slot. Then for all $t \geq 0$

$$D(t) = \min_{0 \leq s \leq t} [A(s) + t - s].$$

Theorem 1: The PF algorithm has the same throughput, i.e., the same average number of packets served per time slot, as an ideal output-queued switch irrespective of the arrival process.

Proof: Without loss of generality, consider only output k and suppose that at time $t = 0$ there are no packets in the queues. Define the following variables:

- $A_k(t)$: cumulative number of packet arrivals up to time t at the inputs of the load-balanced switch destined to output k.
- $B_k(t)$: cumulative number of packet arrivals up to time t at the intermediate inputs of the load-balanced switch destined to output k.
- $C_k(t)$: cumulative number of packet departures up to time t of the load-balanced switch at output k.
- $C_k^{WC}(t)$: cumulative number of packet departures up to time t of a work-conserving server of unit capacity with arrivals given by $B_k(t)$.
- $C_k^{OQ}(t)$: cumulative number of packet departures up to time t at output k of an output-queued switch with arrivals given by $A_k(t)$.
- $q'_k(t)$: queue length at the inputs of the load-balanced switch for packets destined to output k.
- $q''_k(t)$: queue length at the intermediate inputs of the load-balanced switch for packets destined to output k.

Since the cumulative number of departures is always less than or equal to the cumulative number of arrivals, it is easy to see that

$$B_k(t) \geq C_k^{WC}(t). \tag{1}$$

Observe that at any input, from the time instant the last packet in a full frame arrives until the next time the VOQ is considered for scheduling, there can be at most $N - 1$ time slots. It is also

important to notice that at every time slot, only one input can send a packet to the first intermediate input, so the maximum delay a VOQ can experience from the instant it is scheduled to be served until it actually gets service is $N-1$ time slots.

We will now show that the queue length at the inputs is bounded. Note that there can be at most $N(N-1)$ packets in any input without having a full frame. Also, the presence of a full frame ensures that the input will start serving packets (with a delay bounded by $N-1+N-1=2N-2$, as explained in the previous paragraph); thus, at any time instant the maximum number of packets in any input is $N(N-1)+2N-2=N^2+N-2$ (after service) and this number cannot increase any further since we assume that there is at most one arrival per time slot at each input. Hence, all inputs have at most $N(N^2+N-2)=N^3+N^2-2N$ packets destined to output k. Thus

$$q'_k(t)=A_k(t)-B_k(t)\leq N^3+N^2-2N. \tag{2}$$

Now, let us analyze the second switching stage. We will show that if arrivals are given by $B_k(t)$ the difference between cumulative departures in an ideal work-conserving server and the second stage can be bounded; this, in conjuction with (2), will allow us to bound the difference in service between the load-balanced switch implementing PF and the ideal output-queued switch when arrivals are given by $A_k(t)$.

Since we always uniformly send packets to the intermediate inputs, by the pigeonhole principle if the intermediate input queue size is greater than or equal to TN one can only schedule full frames. Thus, if there are TN or more packets destined to output k at the intermediate inputs, then the PF algorithm can only schedule real packets to the intermediate stages. Further, if the number of packets destined to output k is less than or equal to $TN-1$, then each input can send N packets (whether fake or real) over the next N time slots to the intermediate stage. This means that when

$$q''_k(t)\geq N^2+TN$$

the load-balanced switch will only send full frames to the intermediate inputs.

At any time t the load-balanced switch can only be in either of the following two states:

$$q''_k(t)<N^2+TN \tag{3}$$
$$q''_k(t)\geq N^2+TN. \tag{4}$$

For a time $\hat{t}$ such that (3) holds we have the following:

$$C_k(\hat{t})>B_k(\hat{t})-N^2-TN.$$

Hence, from (1), Lemma 1, and (2) we have

$$\begin{aligned}C_k(\hat{t})&\geq C_k^{WC}(\hat{t})-N^2-TN\\&=\min_{0\leq s\leq\hat{t}}\left[B_k(s)+\hat{t}-s\right]-N^2-TN\\&\geq\min_{0\leq s\leq\hat{t}}\left[A_k(s)+\hat{t}-s\right]-N^3-2N^2-(T-2)N\end{aligned} \tag{5}$$

and since an output-queued switch can be modeled as a work-conserving server with unit capacity we have

$$=C_k^{OQ}(\hat{t})-N^3-2N^2-(T-2)N.$$

Thus for (3) we have

$$C_k(\hat{t})\geq C_k^{OQ}(\hat{t})-N^3-2N^2-(T-2)N. \tag{6}$$

The only way to get to state (4) is through (3), where (5) holds for the first time slot when the new state is reached. After that, the switch cannot generate fake packets any longer; it only serves the remaining ones on the queues, which can be at most $(T+N)(N-1)$. Thus the difference in service between our algorithm and the work-conserving server can only increase up to $(T+N)(N-1)$ packets. Therefore, for any time $\tilde{t}$ such that (4) holds, and using Lemma 1 and (2), we have

$$\begin{aligned}C_k(\tilde{t})&\geq C_k^{WC}(\tilde{t})-N^2-TN-(T+N)(N-1)\\&=C_k^{WC}(\tilde{t})-2N^2-T(2N-1)+N\\&=\min_{0\leq s\leq\tilde{t}}\left[B_k(s)+\tilde{t}-s\right]-2N^2-T(2N-1)+N\\&\geq\min_{0\leq s\leq\tilde{t}}\left[A_k(s)+\tilde{t}-s\right]-N^3-3N^2-T(2N-1)\\&\quad+3N\\&=C_k^{OQ}(\tilde{t})-N^3-3N^2-T(2N-1)+3N.\end{aligned}$$

Thus

$$C_k(\tilde{t})\geq C_k^{OQ}(\tilde{t})-N^3-3N^2-T(2N-1)+3N. \tag{7}$$

From (6) and (7), since the number of packets served by PF is within a constant of the number of packets served by an ideal output-queued switch, it follows that PF has the same throughput of the output-queued switch, irrespective of the arrival process. ■

V. Improvements to PF

Although the original algorithm is stable as proved in Section IV-A, it can be further improved. Recall that when fake packets are being sent, they have to be processed at the intermediate inputs as real packets, and discarded only when they arrive at the outputs. The time slots consumed serving fake packets lead to an increase in the average delay of the switch, so it is beneficial to further control the maximum amount of fake packets that are generated.

A first idea is to keep track of the number of fake packets in the intermediate inputs, and when they reach a threshold the algorithm is only allowed to send full frames. The problem with this approach is that the counter which keeps track of fake packets has to be updated up to $2N$ times every time slot, once when a fake packet is being sent to the intermediate inputs and once when the packet is discarded at the outputs. Further, this approach is not convenient under high load, when it is extremely important to minimize any additional processing due to fake packets.

This concern lead us to consider a different approach: keeping track of the number of padded frames at the intermediate inputs destined for every output. Now there will be N counters, one per output, and they will only need to be updated every N time

slots, either during scheduling or at the end of serving a padded frame at the output. Additionally, since the number of padded frames, rather than fake packets, is being upper-bounded, under heavy load the number of fake packets in the intermediate inputs is reduced, because the probability of serving a padded frame with a large number of fake packets decreases.

The idea is that we should only schedule a padded frame destined to output k if the length of the intermediate input VOQs for output k is less than T *and* if the number of padded frames in intermediate inputs that need to be sent to output k is less than the threshold W. Recall that the parameter T in the original algorithm was used to guarantee stability, whereas W is used here to improve packet delay under light load. In Sections VI and VII we will empirically study the tradeoff in both parameters.

This reasoning leads us to the following modifications to the original algorithm. The improved algorithm will now be called PF+.

- Modify step 4) as follows. Assume that at input i input VOQ_{ik} was scheduled to be served. Let PF_k be a counter that keeps track of the number of padded frames in the intermediate inputs destined to output k. Let W be an integer parameter, and let L_k and T be defined as in the original PF algorithm. Then check if $PF_k < W$ *and* if $L_k < T$. If so, then keep the schedule, and update $PF_k \leftarrow PF_k + 1$. Else, do not schedule input VOQ_{ik}.
- Modify step 5) as follows. When serving an input VOQ that was scheduled in step 4), if there is no packet to send, send a fake packet to the intermediate inputs. Otherwise, send the stored packets. During the service time if packets arrive to the input VOQ in such a way that it is not necessary to send fake packets, update counter $PF_k \leftarrow PF_k - 1$.
- Add a new step after step 6): At output k, once it completely receives a padded frame from the intermediate inputs, update $PF_k \leftarrow PF_k - 1$.

To help clarify the algorithm, please refer to Appendix B for a pseudocode description of the changes done to PF in PF+.

Notice that if either $W = 0$ or $T = 0$, then the PF+ algorithm reduces to the UFS algorithm, since no padding is allowed. Hence, PF+ is a generalization of UFS.

A. Stability of PF+

In PF+ we are adding one more constraint to send padded frames, so the number of padded frames sent can only decrease with respect to PF. With that in mind, we state the following theorem.

Theorem 2: The PF+ algorithm has the same throughput (i.e., the same average number of packets served per time slot) as an ideal output-queued switch irrespective of the arrival process.

Proof: The proof follows from the proof for Theorem 1. The only difference is that since now we have one more constraint (W) to schedule padded frames the number of fake packets generated decreases, which in turn means that the difference between the number of packets served by PF+ and the ideal output-queued switch is smaller than the difference in case PF would have been used.

In this case the maximum amount of fake packets in the intermediate inputs can be at most

$$K = \min\{(T+N)(N-1), W(N-1)\}.$$

For a time $\hat{t}$ such that the switch is in state (3), the result still is the same as in Theorem 1

$$C_k(\hat{t}) \geq C_k^{OQ}(\hat{t}) - N^3 - 2N^2 - (T-2)N. \tag{8}$$

And for a time $\tilde{t}$ such that the switch is in state (4), using (2) and Lemma 1, we have the following:

$$\begin{aligned} C_k(\tilde{t}) &\geq C_k^{WC}(\tilde{t}) - N^2 - TN - K \\ &= \min_{0\leq s\leq \tilde{t}} \left[B_k(s) + \tilde{t} - s\right] - N^2 - TN - K \\ &\geq \min_{0\leq s\leq \tilde{t}} \left[A_k(s) + \tilde{t} - s\right] - N^3 - 2N^2 \\ &\quad - (T-2)N - K \\ &= C_k^{OQ}(\tilde{t}) - N^3 - 2N^2 - (T-2)N - K. \end{aligned}$$

Thus

$$C_k(\tilde{t}) \geq C_k^{OQ}(\tilde{t}) - N^3 - 2N^2 - (T-2)N - K. \tag{9}$$

From (8) and (9), since the number of packets served by PF+ is within a constant of the number of packets served by an ideal output-queued switch, it follows that PF+ has the same throughput as that of the output-queued switch, irrespective of the arrival process. ■

VI. Simulations

In our simulations, we use a Bernoulli traffic model, where the matrix of arrival rates at input i for output j $\{\lambda_{ij}\}$ has random entries subject to

$$\sum_j \lambda_{ij} = \lambda \,\forall\, i \qquad \sum_i \lambda_{ij} = \lambda \,\forall\, j.$$

The parameter λ indicates the load on the switch and will be varied to study the performance of the switch at various loads. The unit of time for all simulations is one time slot. All our simulations have been run for 200 000 time slots. Each figure presented is the average of 30 randomly generated runs, where the error bars show the 95% confidence intervals. Our simulations showed that in the case of PF+, the average delay for low load is determined by the most stringent parameter, whether it is W or T: once either threshold is reached the algorithm starts sending only full frames, no matter the state of the other threshold. However, under high load the probability of having high queue sizes at the intermediate inputs increases; thus, the parameter T is the one that affects the performance of the algorithm in this region. In this case W is not that important, since full frames become more likely. Besides, under heavy load it is undesirable to send fake packets, since they increase the average packet delay of the switch, so T acts as a controller that avoids this undesirable effect for heavy traffic loads. As an illustration, in Figs. 3 and 4 we show the performance of PF+ compared with the performance of using only T or W. It can be seen that PF+ behaves better than PF which uses only the T parameter.

Since UFS and FOFF seem to be the most promising algorithms to avoid resequencing in load-balanced switches, we compared PF and PF+ to these two algorithms.

In Figs. 5 and 6 we compare both PF and PF+ against UFS. As can be seen, the improvement under low load is quite significant. This is because UFS can only send packets when there is a full frame, while in our case we can send padded frames. In

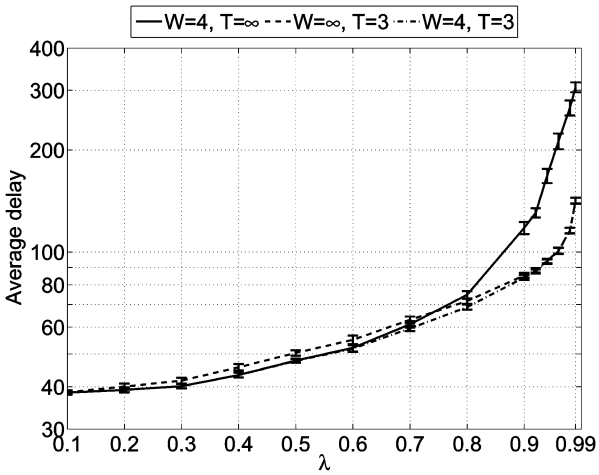

Fig. 3. Performance of PF+ for a 20×20 switch under nonuniform independent Bernoulli arrivals.

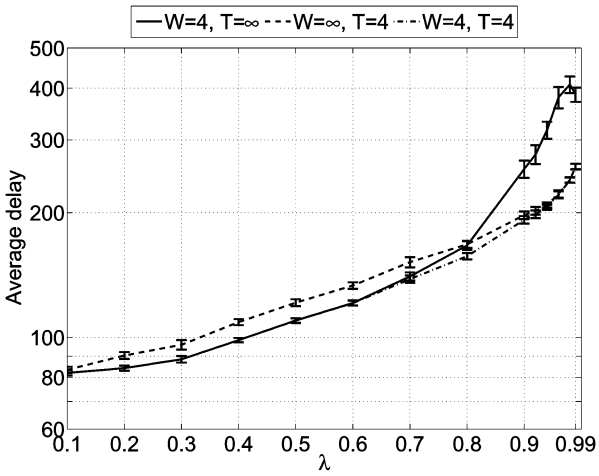

Fig. 4. Performance of PF+ for a 50×50 switch under nonuniform independent Bernoulli arrivals.

Figs. 7 and 8 we compare our algorithms with FOFF. Still our algorithms show a significantly better performance. See Tables I and II for a quantitative comparison. It can be seen in the tables that the improvement can be as high as 49.05% for $N = 50$ and 45.70% for $N = 20$ using PF+.

While programming the FOFF algorithm for our simulations we found out that if a crossbar model is used for the two switching stages, as opposed to the mesh model proposed in [13], one can no longer use FIFO queues for the input buffering in the FOFF. In fact, we noticed that when using a FIFO queue, the throughput of the algorithm seems to be limited to around 2/3 of the maximum possible throughput of the switch. The intuition behind this is that since FOFF keeps track of the last intermediate input the flow from input i to output j sent a packet, it can only start sending packets once the crossbar is in the right configuration. Due to this, it is possible to come up with an adversarial traffic pattern that will make the algorithm always wait for up to $N-1$ time slots before it can start sending packets for the VOQ that is in service, even if you are sending full frames only. This would make the queues at the inputs to grow unbounded.

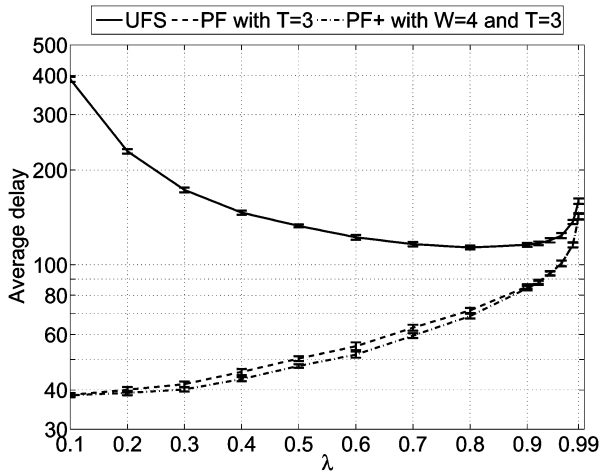

Fig. 5. Average delay comparison against UFS for a 20×20 switch under nonuniform independent Bernoulli arrivals.

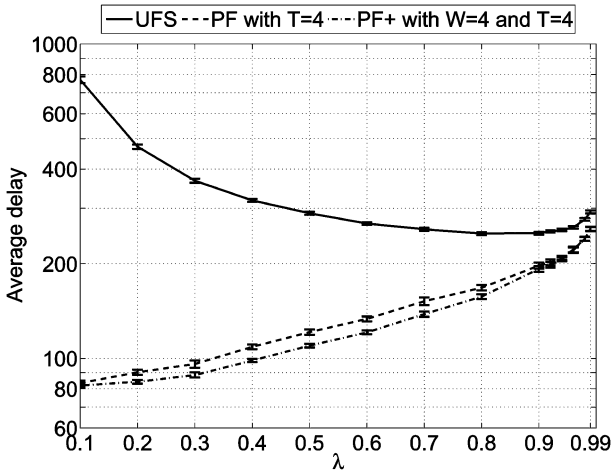

Fig. 6. Average delay comparison against UFS for a 50×50 switch under nonuniform independent Bernoulli arrivals.

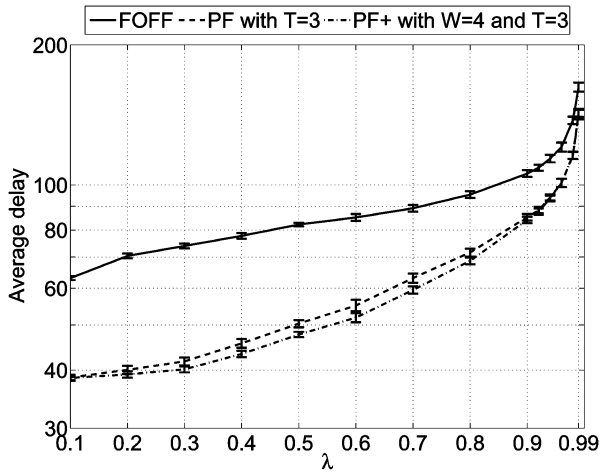

Fig. 7. Average delay comparison against FOFF for a 20×20 switch under nonuniform independent Bernoulli arrivals.

As suggested to us by one of the developers of the FOFF algorithm (Isaac Keslassy), this problem can be solved by using random access memory (RAM)-based queues both at the input

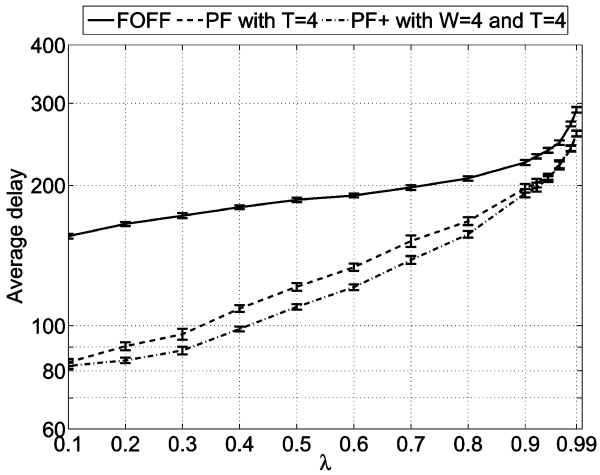

Fig. 8. Average delay comparison against FOFF for a 50×50 switch under nonuniform independent Bernoulli arrivals.

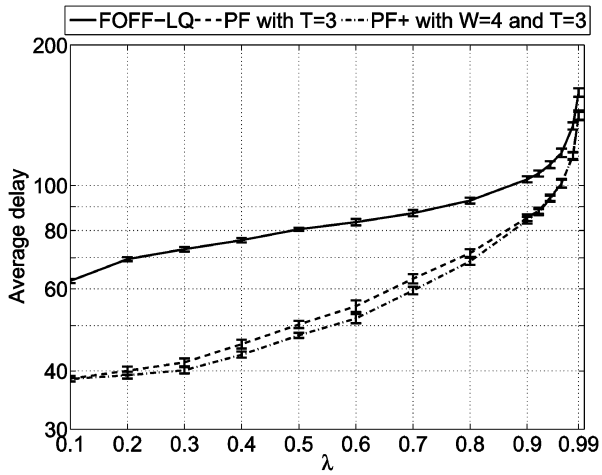

Fig. 9. Average delay comparison against FOFF-LQ for a 20×20 switch under nonuniform independent Bernoulli arrivals.

TABLE I
IMPROVEMENT OF PF AND PF+ COMPARED TO FOFF FOR $N = 20$ UNDER NONUNIFORM INDEPENDENT BERNOULLI ARRIVALS

Load	FOFF	PF	PF+	% PF	% PF+
0.10	63.05	38.57	38.46	38.83%	38.99%
0.20	70.37	40.04	39.16	43.10%	44.36%
0.30	73.93	41.76	40.14	43.52%	45.70%
0.40	77.73	45.66	43.32	41.25%	44.26%
0.50	82.21	50.34	47.69	38.76%	41.99%
0.60	85.15	55.07	51.86	35.33%	39.10%
0.70	89.12	63.08	59.48	29.21%	33.25%
0.80	95.35	71.61	68.70	24.90%	27.95%
0.90	105.78	85.22	84.10	19.43%	20.50%
0.92	108.97	88.28	87.64	18.99%	19.58%
0.94	113.91	94.02	93.54	17.46%	17.88%
0.96	120.66	101.02	100.98	16.27%	16.31%
0.98	137.73	115.88	115.70	15.86%	15.99%
0.99	162.36	114.90	141.73	11.98%	12.71%

TABLE II
IMPROVEMENT OF PF AND PF+ COMPARED TO FOFF FOR $N = 50$ UNDER NONUNIFORM INDEPENDENT BERNOULLI ARRIVALS

Load	FOFF	PF	PF+	% PF	% PF+
0.10	155.55	83.42	82.00	46.37%	47.28%
0.20	165.29	90.36	84.20	45.33%	49.05%
0.30	172.12	95.90	88.48	44.28%	48.59%
0.40	179.51	108.84	98.40	39.37%	45.18%
0.50	186.19	121.15	109.74	34.93%	41.06%
0.60	190.21	133.45	120.96	29.84%	36.41%
0.70	197.92	151.90	138.32	23.25%	30.11%
0.80	206.99	167.56	157.04	19.05%	24.13%
0.90	223.88	197.14	192.31	11.94%	14.10%
0.92	230.90	202.10	198.35	12.47%	14.10%
0.94	237.78	208.62	206.25	12.26%	13.26%
0.96	247.02	222.02	220.66	10.12%	10.67%
0.98	270.73	240.45	239.50	11.18%	11.54%
0.99	290.43	258.31	258.25	11.06%	11.08%

and output. To be fair, we included this modification to FOFF in our simulations. It is important to note that our simulations were not run using the mesh model for the two switching stages of the load-balanced switch, since we only focused on the crossbar model.

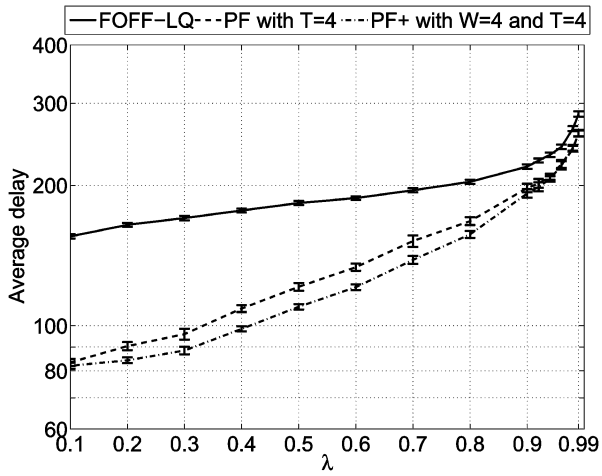

Fig. 10. Average delay comparison against FOFF-LQ for a 50×50 switch under nonuniform independent Bernoulli arrivals.

Additionally, we explore the question whether FOFF's performance can be improved dramatically compared to PF+ by choosing nonfull frames to serve using queue length information. Recall that PF+ selects the longest queue to serve when a full frame is not available. Thus, it is possible that the performance of PF+ is primarily due to the longest queue first policy, and not due to the use of padded frames. In Figs. 9 and 10 we compare the performance of FOFF with longest queue first (that we will call FOFF-LQ) against PF+. We see that PF+ continues to perform better, indicating that the use of padded frames is important to achieve good performance. In other words, eliminating the reordering buffer in FOFF is evidently critical to improve performance.

Finally, we compare the load-balanced switch with PF+ against another architecture for the input-queued switch and the ideal output-queued switch. For the input-queued switch we decided to compare our algorithm to iSLIP, which is known to be easy to implement and has been successfully deployed

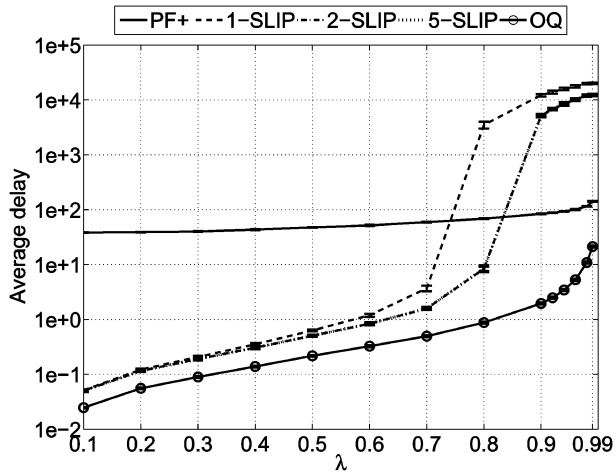

Fig. 11. Average delay comparison against iSLIP for a 20×20 switch under nonuniform independent Bernoulli arrivals.

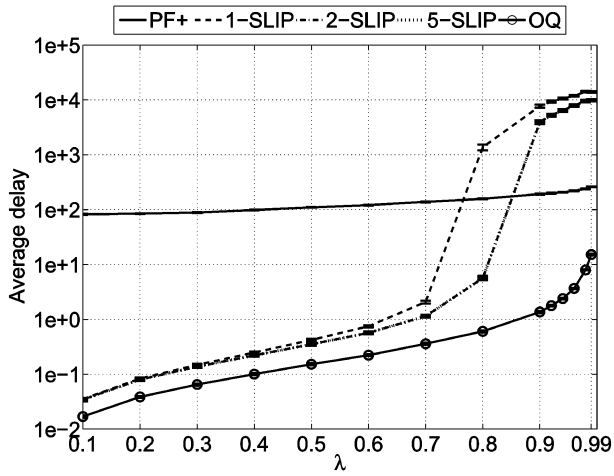

Fig. 12. Average delay comparison against iSLIP for a 50×50 switch under nonuniform independent Bernoulli arrivals.

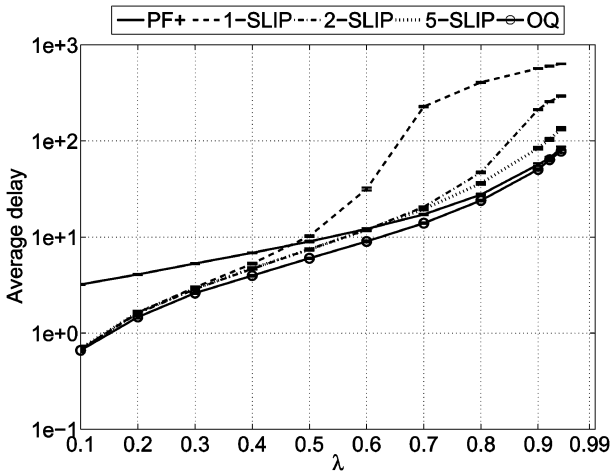

Fig. 13. Average delay comparison against iSLIP for a 20×20 switch under bursty arrivals.

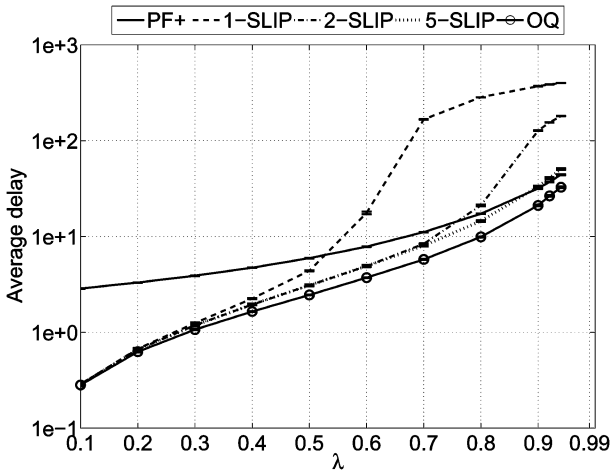

Fig. 14. Average delay comparison against iSLIP for a 50×50 switch under bursty arrivals.

commercially. (For the convenience of the reader, we provide a brief description of the iSLIP algorithm in Appendix C.) In [4] it is proven that iSLIP converges in at most N iterations, where N is the switch size, but it is stated that through simulations they have found that usually the expected number of iterations needed to converge is less than or equal to $\log_2 N$. For our simulations, we compare our algorithm against 1-SLIP, 2-SLIP, and 5-SLIP in the case when $N = 20$ and $N = 50$.

In Figs. 11 and 12, where we used nonuniform independent Bernoulli arrivals, it can be observed that although in the low-load region PF+ performs poorly compared to both iSLIP and the output-queued switch, in the high load region our algorithm outperforms iSLIP, showing that our algorithm tends to be more stable under high loads.

In [4] it is stated that a more realistic traffic model is one with bursty arrivals. This is due to the fact that real network traffic is highly correlated and packets tend to arrive in bursts, corresponding to packetized files. So we decided to compare the two algorithms, using the same bursty traffic model used in [4]. Packets are generated by an on–off arrival process modulated by a two-state Markov chain such that at every input the source generates a busy period of packets (with the same destination) followed by a period of inactivity. The number of packets in the busy and idle periods are geometrically distributed, and the destination for a burst is uniformly distributed over all outputs.

In Figs. 13 and 14 we present the result of the simulations, where the mean of the busy periods is 128 packets. Although not shown here, we also run simulations with busy period means of 16, 32, and 64 packets. We were able to observe that the bigger the bursts the closer PF+ approaches the output-queued average delay, especially at higher loads. It should be noted that in high loads PF+ still outperforms iSLIP, although not as dramatically as in the case of Bernoulli arrivals. We point out that PF+ is provably stable at all loads whereas iSLIP can not achieve 100% throughput (the reader is referred to [9] for an example showing that iSLIP cannot achieve 100% throughput). We also run simulations of both UFS and FOFF under this traffic model and our algorithm continued to have a better average delay than these two algorithms. These results are not shown here since the figures are quite repetitive. It is interesting to note that our algorithm has a performance similar to that of the output-queued switch for bursty arrivals. This is due to the fact that under bursty

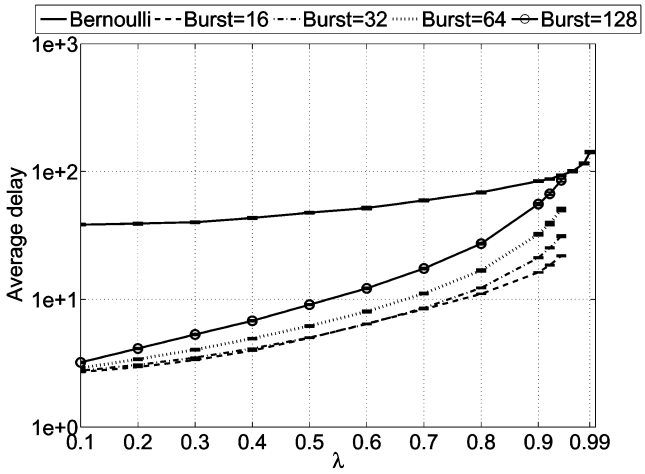

Fig. 15. Variation of average delay of PF+ for a 20 × 20 switch under different traffic models.

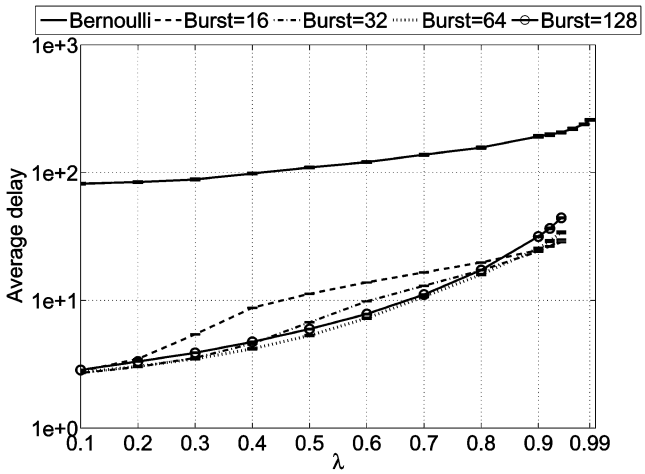

Fig. 16. Variation of average delay of PF+ for a 50 × 50 switch under different traffic models.

traffic the algorithm does not generate as many fake packets as under Bernoulli traffic, so the performance penalty incurred decreases substantially, as can be seen in Figs. 15 and 16.

VII. Implementation Complexity

Let us first analyze the PF algorithm. It must be noted that step 1) requires storing a packet in its input VOQ, which takes $O(1)$ time. Steps 2), 5), and 6) do not differ significantly from the original load-balanced switch, which runs in $O(1)$ time [9].

Steps 3) and 4) need a more careful analysis though. The round-robin search in step 3) can be implemented in $O(\log N)$ time [17]. In step 4) we first need to find the largest nonempty queue in a given input. This can be done by using a comparison tree circuit where the leaf nodes indicate the queue lengths and the root indicates the ID of the largest one, as depicted in Fig. 17. This design incurs $O(\log N)$ gate delays. Next we have to determine whether the threshold $L_k < T$ has been reached or not. We stress that we do not need to know the value of the L_k variable but only if the threshold has been reached. Furthermore, only intermediate input 1 updates the value of L_k, so all inputs only require a read access. Thus the communication between linecards only requires a shared bus used to broadcast a N-bit vector indicating whether the thresholds have been reached or not. Thus this concurrent read requires only $O(1)$ time. We also highlight that since only one update is required per frame, communication between linecards occurs at a $1/N$th of the line rate.

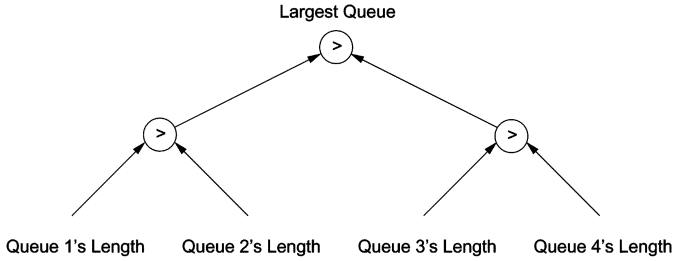

Fig. 17. Example of a comparison tree when $N = 4$. The root node indicates the ID of the largest queue.

TABLE III
Variation of Optimal T and W for a Given N Under Uniform i.i.d. Bernoulli Arrivals

N	5	10	20	30	40	50
T	3	3	3	4	4	4
W	4	4	4	4	4	4

PF+ is similar, but with the difference that now we have an additional threshold W and N counters (PF_k) that need to be increased/decreased by the inputs and outputs. For the threshold W, a solution similar to the one found for T can be used, so we only need to focus on how to share and update the counters. An option is to implement each counter in its respective output and let the output do the comparison $PF_k < W$ and share with each input the one-bit result with a shared bus similar to the one proposed previously for T and W. Each input only needs to convey a two-bit vector to its output indicating whether the counter should be increased, decreased, or left untouched. Note that since scheduling is done once every frame, communication between linecards occurs at a $1/N$th of the line rate.

For our algorithms we need to generate fake packets at the inputs, and the outputs must be able to differentiate between real and fake ones, since fake packets must be discarded. In this work we assumed that all incoming packets are segmented into fixed-size cells, and then reassembled at the outputs before leaving the switch. One easy way to distinguish between fake and real cells is to include a bit in the header of the segmented cells, where "1" could mean "real." The generated fake packets would be branded "0" meaning "fake." Thus, the outputs only have to discard packets that have a bit header of "0" and reassemble the remaining ones.

One interesting question is how the optimal parameters T and W vary when you change N. In particular, if T and W are very sensitive to either N or the traffic load model, then the algorithm would be difficult to implement in practice. This is important since a real switch may not have a static configuration, but instead network providers must be able to increase or decrease the number of line cards according to their needs. Table III shows the optimal parameters for different values of N. To find these

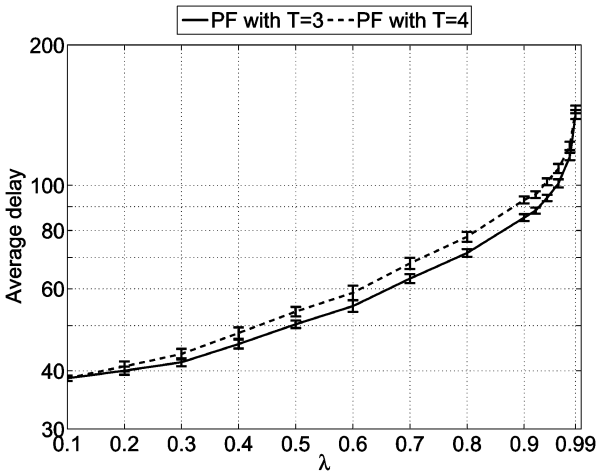

Fig. 18. Average delay for a 20×20 switch using PF with $T = 3$ and $T = 4$ under nonuniform independent Bernoulli arrivals.

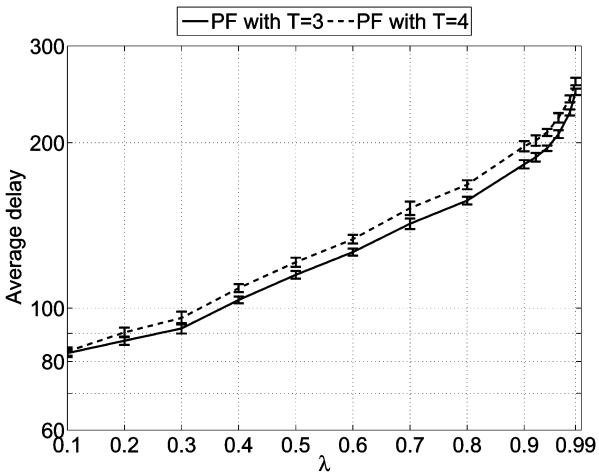

Fig. 19. Average delay for a 50×50 switch using PF with $T = 3$ and $T = 4$ under nonuniform independent Bernoulli arrivals.

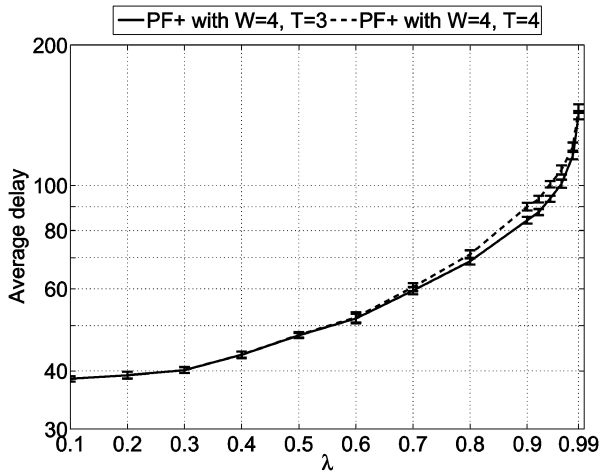

Fig. 20. Average delay for a 20×20 switch using PF+ with $T = 3$ and $T = 4$ under nonuniform independent Bernoulli arrivals.

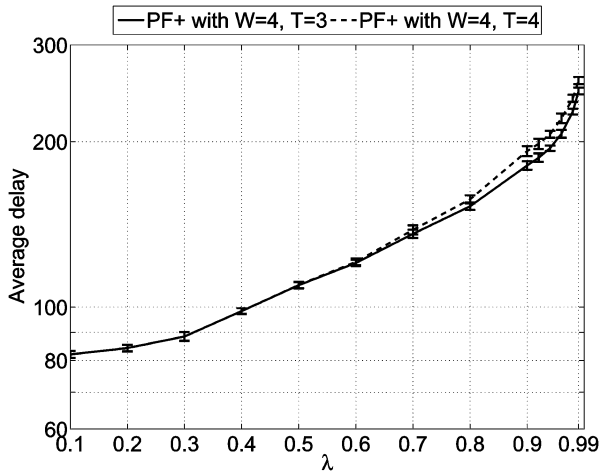

Fig. 21. Average delay for a 50×50 switch using PF+ with $T = 3$ and $T = 4$ under nonuniform independent Bernoulli arrivals.

values we used a Bernoulli traffic model, where the arrival rate matrix $\{\lambda_{ij}\}$ has uniform entries such that

$$\lambda_{ij} = \frac{\lambda}{N} \; \forall \, i \text{ and } \forall \, j.$$

In extensive simulations, we found that the optimal choice of T and W for the uniform i.i.d. Bernoulli traffic model seems to perform well for nonuniform Bernoulli loads and bursty arrivals as well, and the optimal parameter only changes slightly.

As can be seen, W remains constant for a large range of N, while T changes slightly. However, if you use a suboptimal value of T the change in the performance is not dramatic, especially for PF+ as can be seen in Figs. 18–21, where the suboptimal parameter $T = 3$ for $N = 50$ becomes optimal under nonuniform Bernoulli arrivals. This means that an Original Equipment Manufacturer (OEM) implementing our algorithms has at least two choices: keep the parameters constant to avoid reconfiguration, or make changes if N changes dramatically.

Although surprising at first sight, the relatively small changes in T and W are due to the fact that they are thresholds for *frames* composed of N *packets*. Hence, although these parameters are nearly constant over large variations in N and the traffic load, the threshold in terms of the number of packets changes.

We summarize the implementation characteristcs of PF+ below.

i) PF+ is easy to implement, requires only FIFO queues, and does not need reordering buffers at the output ports.

ii) One may argue that a possible disadvantage is that one has to choose the parameters T and W to ensure good performance. For example, $T = W = 0$ reduces PF+ to UFS, which has poor performance at low loads. On the other hand, we have shown that T and W are quite insensitive to the switch size N and the traffic model. Thus, having to choose these parameters does not seem to be a significant issue.

VIII. Conclusions and Future Directions

In this paper we have presented Padded Frames, a novel scheduling algorithm for load-balanced Birkhoff-von Neumann switching architectures. Through the insertion of fake packets into the intermediate input queues, PF prevents packet reordering and starvation of lightly loaded queues. We also

presented PF+, a modification of the original algorithm that improves the average packet delay while maintaining the stability and throughput guarantees of PF.

The algorithm requires no speedup and is decentralized. In addition, we showed that the information that needs to be shared between line cards is not going to have a significant impact on the implementation of high-speed switches since it is minimal and exchanged at a $1/N$th of the line rate.

We have shown, by means of both analysis and simulation, that both PF and PF+ are stable for *any* admissible traffic patterns, and their performance is significantly better than those of existing algorithms such as Uniform Frame Spreading and Full Ordered Frames First.

We also compared our algorithm against the input-queued switch implemented with iSLIP and the ideal output-queued switch. From our results we showed that our algorithm outperforms iSLIP under heavy traffic loads, and that in the case of a traffic model with bursty arrivals our algorithm approaches the average delay performance of the output-queued switch.

In the present work we focused on proving the stability of our algorithms, and the next step would be to present a theoretical analysis of the delay performance of the algorithm. It would also be interesting to show analytically that the optimal values of both T and W are relatively insensitive to the switch parameters.

Appendix I
Pseudocode Description of PF

```
function ProcessIncomingPacket (Packet)
    Input ← FindInput (Packet)
    Output ← FindOutput (Packet)
    PushInputVOQ (Input, Output, Packet)
return
function ScheduleInputVOQ (Time, Priority)
    if (Time mod N) == 0 then
        for Input in 1 to N do
            for i in 0 to N − 1 do
                Output ← Priority [Input] +i
                /* Length of unscheduled packets */
                Length ← InputVOQ (Input, Output)
                if Length ≥ N then
                    ServeInputVOQ [Input] ← Output
                    Priority [Input] ← Output +1
                    Scheduled [Input] ← True
                    break
        for Input in 1 to N do
            if Scheduled [Input] == False then
                Output ← LargestInputVOQ (Input)
                if Output ≠ ∅ then
                    /* Length of VOQ at intermediate input 1 */
                    Length ← IntInput1VOQ (Output)
                    if Length < T then
                        ServeInputVOQ [Input] ← Output
                        Scheduled [Input] ← True
    else
        return
return ServeInputVOQ, Priority, Scheduled
```

```
function FirstStageSwitch (Time)
    for Input in 1 to N do
        if Scheduled [Input] == True then
            Output ← ServeInputVOQ [Input]
            IntInput ← [ (Input + Time −1 ) mod N ] +1
            if SeqNo [Input] == IntInput then
                P ← PopInputVOQ (Input, Output)
                if P == ∅ then
                    P ← FakePacket
                PushIntInputVOQ (IntInput, Output, P)
                SeqNo [Input] ← (SeqNo [Input] +1) mod N
                if SeqNo [Input] == 0 then
                    SeqNo [Input] ← N
return

function SecondStageSwitch (Time)
    for Output in 1 to N do
        IntInput ←[ (Output + Time −2) mod N] +1
        Packet ← PopIntInputVOQ (IntInput, Output)
        if Packet ≠ ∅ and Packet ≠ FakePacket then
            Departing (Packet, Output)
return
```

Appendix II
Pseudocode Description of PF+

The following pseudocode summarizes the changes implemented in PF+:

```
function ScheduleInputVOQ(Time, Priority)
    if (Time mod N) == 0 then
        for Input in 1 to N do
            for i in 0 to N − 1 do
                Output ← Priority[Input] +i
                /* Length of unscheduled packets */
                Length ← InputVOQ(Input, Output)
                if Length ≥ N then
                    ServeInputVOQ[Input] ← Output
                    Priority[Input] ← Output +1
                    Scheduled[Input] ← True
                    break
        for Input in 1 to N do
            if Scheduled[Input] == False then
                Output ← LargestInputVOQ(Input)
                if Output ≠ ∅ then
                    /* Length of VOQ at intermediate input 1 */
                    Length ← IntInput1VOQ(Output)
                    /* Start of change */
                    NoPF ← IntInputPF(Output)
                    if Length < T and NoPF < W then
                        NoPF ← NoPF +1
                        UpdateIntInputPF(Output, NoPF)
                        /* End of change */
                        ServeInputVOQ[Input] ← Output
                        Scheduled[Input] ← True
    else
        return
return ServeInputVOQ, Priority, Scheduled
```

```
function FirstStageSwitch(Time)
    for Input in 1 to N do
        if Scheduled[Input] == True then
            Output ← ServeInputVOQ[Input]
            IntInput ← [(Input + Time − 1) mod N] + 1
            if SeqNo[Input] == IntInput then
                P ← PopInputVOQ(Input, Output)
                if P == ∅ then
                    P ← FakePacket
                    /* Start of change */
                    FkPkt ← True
                    /* End of change */
                PushIntInputVOQ(IntInput, Output, P)
                SeqNo[Input] ← (SeqNo[Input] + 1) mod N
                if SeqNo[Input] == 0 then
                    SeqNo[Input] ← N
                    /* Start of change */
                    if SendingPaddedFrame() == True then
                        if FkPkt == False then
                            NoPF ← IntInputPF(Output)
                            NoPF ← NoPF − 1
                            UpdateIntInputPF(Output, NoPF)
                        else
                            FkPkt ← False
                    /* End of change */
return

function SecondStageSwitch(Time)
    for Output in 1 to N do
        IntInput ← [(Output + Time − 2) mod N] + 1
        Packet ← PopIntInputVOQ(IntInput, Output)
        /* Start of change */
        if Packet == FakePacket then
            PaddedFrame ← True
        /* End of change */
        if Packet ≠ ∅ and Packet ≠ FakePacket then
            Departing(Packet, Output)
        /* Start of change */
        if IntInput == N and PaddedFrame == True then
            PaddedFrame ← False
            NoPF ← IntInputPF(Output)
            NoPF ← NoPF − 1
            UpdateIntInputPF(Output, NoPF)
        /* End of change */
return
```

APPENDIX III
DESCRIPTION OF iSLIP

Here we present the basic form of the iSLIP algorithm as presented in [4].

1) *Request.* Each unmatched input sends a request to every output for which it has a queued cell.
2) *Grant.* If an unmatched output receives any requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the granted input if and only if the grant is accepted in step 3) of the *first iteration*.
3) *Accept.* If an unmatched input receives a grant, it accepts the one that appears next in a fixed round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output if and only if the acceptance is given in the *first iteration*.

Through several iterations of the steps described above, the algorithm tries to find a maximal matching for every time slot: the largest match without removing the connections found in previous iterations. The number of iterations then determines the name of the algorithm, e.g., if you use iSLIP with two iterations you call it 2-SLIP.

ACKNOWLEDGMENT

The authors would like to thank Isaac Keslassy for answering many of their questions on the FOFF algorithm, and the anonymous reviewers for their valuable comments on improving the quality of the paper, especially the suggestions regarding how to efficiently implement the algorithm.

REFERENCES

[1] N. McKeown, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," in *Proc. IEEE INFOCOM 1996*, San Francisco, CA, Mar. 1996, vol. 1, pp. 296–302.
[2] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch (extended version)," *IEEE Trans. Commun.*, vol. 47, pp. 1260–1267, Aug. 1999.
[3] H. N. Gabow and R. E. Tarjan, "Faster scaling algorithms for network problems," *SIAM J. Comput.*, vol. 18, pp. 1013–1036, 1989.
[4] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Netw.*, vol. 7, no. 2, pp. 188–201, Apr. 1999.
[5] L. Tassiulas, "Linear complexity algorithms for maximum throughput in radio networks and input-queued switches," in *Proc. IEEE INFOCOM 1998*, San Francisco, CA, Mar. 1998, pp. 533–539.
[6] P. Giaccone, D. Shah, and B. Prabhakar, "An implementable parallel scheduler for input-queued switches," *IEEE Micro*, vol. 22, no. 1, pp. 19–25, Jan. 2002.
[7] A. Bianco, M. Franceschinis, S. Ghisolfi, A. M. Hill, E. Leonardi, F. Neri, and R. Webb, "Frame-based matching algorithms for input-queued switches," in *Proc. High Performance Switching and Routing, HPSR 2002*, Kobe, Japan, May 2002, pp. 69–76.
[8] M. J. Neely and E. Modiano, "Logarithmic delay for $N \times N$ packet switches," in *Proc. High Performance Switching and Routing, HPSR 2004*, Phoenix, AZ, Apr. 2004, pp. 1–7.
[9] C. S. Chang, D. S. Lee, and Y. S. Jou, "Load balanced Birkhoff–von Neumann switches, part I: One-stage buffering," *Comput. Commun.*, vol. 25, no. 6, pp. 611–622, 2002.
[10] C. S. Chang, D. S. Lee, and C. M. Lien, "Load balanced Birkhoff-von Neumann switches, part II: Multi-stage buffering," *Comput. Commun.*, vol. 25, no. 6, pp. 623–634, 2002.
[11] I. Keslassy, S.-T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, and N. McKeown, "Scaling internet routers using optics," in *Proc. ACM SIGCOMM 2003*, Karlsruhe, Germany, Aug. 2003.
[12] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Trans. Netw.*, vol. 2, no. 1, pp. 1–15, Feb. 1994.
[13] I. Keslassy, "The load-balanced router," Ph.D. dissertation, Stanford Univ., Stanford, CA, 2004.
[14] C. S. Chang, D. S. Lee, and C. Y. Yue, "Providing guaranteed rate services in the load balanced Birkhoff-von Neumann switches," in *Proc. IEEE INFOCOM 2003*, San Francisco, CA, 2003, pp. 1622–1632.

[15] I. Keslassy and N. McKewon, "Maintaining packet order in two-stage switches," in *Proc. IEEE INFOCOM 2002*, New York, NY, Jun. 2002, pp. 1032–1041.
[16] C. S. Chang, *Performance Guarantees in Communication Networks*. New York: Springer-Verlag, 2000.
[17] P. Gupta and N. McKeown, "Designing and implementing a fast crossbar scheduler," *IEEE Micro*, vol. 19, no. 1, pp. 20–28, Jan./Feb. 1999.

Juan José Jaramillo (S'06) received the B.S. degree (*summa cum laude*) in electronics engineering from the Universidad Pontificia Bolivariana, Medellin, Colombia, in 1998, and the M.S. degree in electrical engineering from the University of Illinois at Urbana Champaign in 2005, where he is currently pursuing the Ph.D. degree.

From 1999 to 2003, he worked at Empresas Publicas de Medellin in Colombia. He is the former recipient of a Fulbright fellowship. His research interests include communication networks and game theory.

Fabio Milan (S'05) received the M.S. degree in telecommunication engineering and the Ph.D. degree in electronic and communication engineering, both from Politecnico di Torino, Turin, Italy, in 2003 and 2007, respectively.

Between September 2000 and July 2001, he was an Exchange Student at the Universidad Publica de Navarra, Pamplona, Spain. Between June 2005 and May 2006, he was a Visiting Scholar at the University of Illinois at Urbana-Champaign. His research interests include scheduling algorithms for input-queued switches, resource sharing in peer-to-peer networks, and incentives for cooperation in multihop wireless networks.

R. Srikant (S'90–M'91–SM'01–F'06) received the B.Tech. degree from the Indian Institute of Technology, Madras, in 1985, and the M.S. and Ph.D. degrees from the University of Illinois at Urbana-Champaign in 1988 and 1991, respectively, all in electrical engineering.

He was a Member of the Technical Staff at AT&T Bell Laboratories from 1991 to 1995. He is currently with the University of Illinois at Urbana-Champaign, where he is a Professor in the Department of Electrical and Computer Engineering, and a Research Professor in the Coordinated Science Laboratory. His research interests include communication networks, stochastic processes, queueing theory, information theory, and game theory.

Dr. Srikant has served as an Associate Editor of the IEEE TRANSACTIONS ON AUTOMATIC CONTROL, and is currently an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING. He has also served on the editorial boards of special issues of the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS and IEEE TRANSACTIONS ON INFORMATION THEORY. He was an Associate Editor of *Automatica*. He was the Chair of the 2002 IEEE Computer Communications Workshop in Santa Fe, NM, and was a Program Co-Chair of IEEE INFOCOM 2007.