Randomized Scheduling Algorithms for High-Aggregate Bandwidth Switches

Paolo Giaccone, *Member, IEEE*, Balaji Prabhakar, *Member, IEEE*, and Devavrat Shah

***Abstract*—The aggregate bandwidth of a switch is its port count multiplied by its operating line rate. We consider switches with high-aggregate bandwidths; for example, a 30-port switch operating at 40 Gb/s or a 1000-port switch operating at 1 Gb/s. Designing high-performance schedulers for such switches with input queues is a challenging problem for the following reasons: 1) high performance requires finding good matchings; 2) good matchings take time to find; and 3) in high-aggregate bandwidth switches there is either too little time (due to high line rates) or there is too much work to do (due to a high port count).**

We exploit the following features of the switching problem to devise simple-to-implement, high-performance schedulers for high-aggregate bandwidth switches: 1) the state of the switch (carried in the lengths of its queues) changes slowly with time, implying that heavy matchings will likely stay heavy over a period of time and 2) observing arriving packets will convey useful information about the state of the switch. The above features are exploited using hardware parallelism and randomization to yield three scheduling algorithms—APSARA, LAURA, and SERENA. These algorithms are shown to achieve 100% throughput and simulations show that their delay performance is quite close to that of the maximum weight matching, even when the traffic is correlated. We also consider the stability property of these algorithms under generic admissible traffic using the fluid-model technique.

The main contribution of this paper is a suite of simple to implement, high-performance scheduling algorithms for input-queued switches. We exploit a novel operation, called MERGE, which combines the edges of two matchings to produce a heavier match, and study of the properties of this operation via simulations and theory. The stability proof of the randomized algorithms we present involves a derandomization procedure and uses methods which may have wider applicability.

***Index Terms*—Input queued switch scheduling, packet switching, randomized scheduling algorithms.**

I. Introduction

OVER THE past few years the input-queued switch architecture has become dominant in high-speed switching. This is mainly due to the fact that the memory bandwidth of its packet buffers is very low compared with that of an output-queued or a shared-memory architecture.

Fig. 1 shows the logical structure for an input-queued (IQ) switch. Suppose that time is slotted so that at most one packet can arrive at each input in one time slot. Packets arriving at input i and destined for output j are buffered in a "virtual output queue" (VOQ), denoted here by VOQ$_{ij}$. The use of VOQs avoids performance degradation due to the head-of-line blocking phenomenon [2]. Let the average cell arrival rate at input i for output j be λ_{ij}. The incoming traffic is called *admissible* if $\sum_{i=1}^{N} \lambda_{ij} < 1$, and $\sum_{j=1}^{N} \lambda_{ij} < 1$. We assume that packets are switched from inputs to outputs by a crossbar fabric. When switching unicast traffic,[1] this fabric imposes the following constraint: in each time slot, at most one packet may be removed from each input and at most one packet may be transferred to each output.

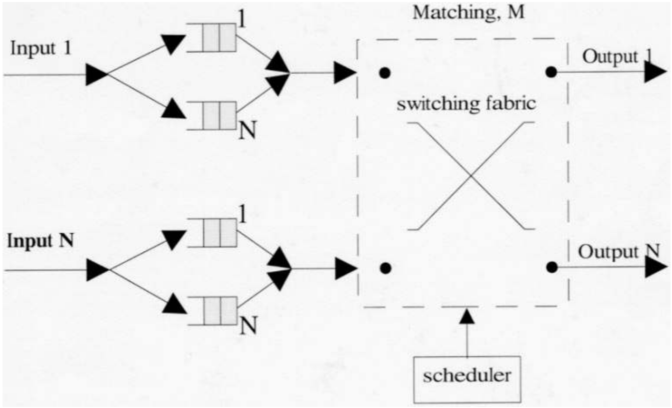

Fig. 1. Logical structure of an input-queued cell switch.

To perform well, an $N \times N$ input-queued switch requires a good packet scheduling algorithm for determining which inputs to connect with which outputs in each time slot. It is well-known that the crossbar constraint makes the switch scheduling problem a matching problem in an $N \times N$ weighted bipartite graph. The weight of the edge connecting input i to output j is often chosen to be some quantity that indicates the level of congestion; for example, queue lengths or the ages of packets.

A matching for this bipartite graph is a valid schedule for the switch. Note that a valid matching can be seen as a permutation of the N outputs. In this paper, we will use the words *schedule*, *matching* and *permutation* interchangeably. A matching of particular importance for this paper is the maximum weight matching (MWM) algorithm. Given a weighted bipartite graph, the MWM finds that matching whose weight is the highest. For example, Fig. 2 shows a weighted bipartite graph and one valid schedule (or matching). We shall use $S(t)$ to denote the schedule used by the switch at time t.

Manuscript received August 8, 2002; revised January 24, 2003. This paper was presented at IEEE INFOCOM 2002, New York, NY, June 2002.

P. Giaccone is with the Dipartimento di Elettronica, Politecnico di Torino, Torino 10129, Italy (e-mail: giaccone@polito.it).

B. Prabhakar is with the Department of Computer Science and Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: balaji@isl.stanford.edu).

D. Shah is with the Department of Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: devavrat@cs.stanford.edu).

Digital Object Identifier 10.1109/JSAC.2003.810496

[1] We do not consider multicast traffic in this paper.

0733-8716/03$17.00 © 2003 IEEE

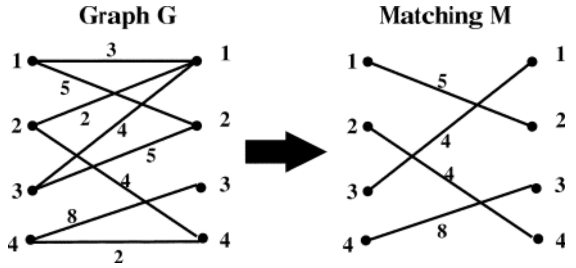

Fig. 2. Example of weighted bipartite graph and its maximum weight matching.

This paper is primarily concerned with designing schedulers for “high-aggregate bandwidth” switches. The aggregate bandwidth of an $N \times N$ switch running at a line rate of L bits/s is defined to be the product NL bits/s. Thus, high-aggregate bandwidth switches can be designed in two ways: a small number of ports (small N) connected to very high-speed lines (large L) and a large number of ports (large N) connected to slower lines (small L). As discussed in [13], the former type of switch typically resides in a “core router,” interconnecting a small number of enterprise networks via high-speed lines. The latter type of switch resides in an “edge router,” which typically has a large number of ports running at relatively lower speeds.

There are two main quantities for measuring the performance of a switch scheduling algorithm: throughput and delay. Early theoretical work on packet switches has been concerned with designing algorithms that achieve 100% throughput. Such algorithms are referred to as “stable” algorithms. In particular, the papers [16], [28], showed that under Bernoulli independent and identically distributed (i.i.d.) packet arrival processes the MWM is stable as long as no input or output is oversubscribed.[2] More recently, other algorithms have been proposed for providing exact delay bounds [4], [14], [24]. Those algorithms in fact provide something much stronger: they allow a switch whose fabric runs at a speedup of between two and four to exactly emulate an output-queued switch. Thus, they are stable and permit the use of sophisticated algorithms for supporting quality-of-service (QoS).

But, all of the above algorithms are too complicated for implementation in high-aggregate bandwidth switches. They require too many iterations (for example, the MWM requires $O(N^3)$ iterations in the worst case), and the computation of weights used in the algorithms of [4], [14], and [24] requires too much information to be communicated between inputs and outputs.

Implementation considerations have, therefore, seen the proposal of a number of practicable scheduling algorithms; notably, iSLIP [18], iLQF [17], RPA [1], MUCS [6], and WFA [26]. However, these algorithms perform poorly compared with MWM when the input traffic is nonuniform: they induce very large delays and their throughput can be less than 100%.

More recently, some particularly simple-to-implement scheduling algorithms have been proposed in [3] and [12] and proven to be stable. But, [3] introduces an extra packet resequencing problem and [12] needs multiple switching fabrics. Nevertheless, these algorithms make a significant point: delivering 100% throughput does not complicate the scheduling problem.

On the other hand, in order to keep delays small, it seems necessary to find good matchings; and finding good matchings takes many iterations and consumes time. And high-aggregate bandwidth switches do not leave much time for scheduling, because they are either connected to very-high-speed lines or they have too many ports.

Our goal of designing simple-to-implement, high-performance schedulers for high-aggregate bandwidth switches leads to the following question: Is it possible for an algorithm to compete with the throughput and delay performance of MWM and yet be simple to implement? If yes, what feature of the scheduling problem remains to be exploited?

The answer lies in recognizing two features of the high-speed switch scheduling problem. 1) *Using memory:* Note that packets arrive (depart) at most one per input (output) per time slot. This means that queue lengths, taken to be the weights by MWM, change very little during successive time slots. Thus, a heavy matching will continue to be heavy over a few time slots, suggesting that carrying some information, or retaining memory, between iterations should help simplify the implementation while maintaining a high level of performance. 2) *Using arrivals:* Since the increase in queue lengths is entirely due to arrivals, it might help to use a knowledge of recent arrivals in finding a matching.

We shall see that both these features considerably simplify the implementation and provide a high performance. We also use some novel techniques for simplifying the implementation.

1) *Hardware parallelism:* Finding heavy matchings essentially involves a search procedure, requiring a comparison of the weight of several matchings. In Section III-A, we propose an algorithm called APSARA, that exploits a natural structure on the space of matchings and uses parallelism in hardware to conduct this search efficiently. In particular, it requires a *single iteration*, is stable, and its delay is comparable to that of MWM.
2) *Randomization:* In a variety of situations where the scalability of deterministic algorithms is poor, randomized algorithms are easier to implement and provide a surprisingly good performance. The main idea is simply stated: Basing decisions upon a few random samples of a large state space is often a good surrogate for making decisions with complete knowledge of the state. See [21] for a general exposition of randomized algorithms, [11], [27] for application to switching, and [20], [23] for other applications to networking. The randomized algorithms in this paper build on the previous work of Tassiulas [27] to a large degree.

A. *Organization of the Paper*

The rest of the paper exploits the above observations and proposes some new algorithms and proof techniques. The results are divided into two parts. Section II deals with throughput and Section III deals with delay. Section II begins by establishing

[2] The weights were taken to be the length of Q_{ij} originally and later work [19] took the weights to be the age of the oldest packet in Q_{ij}.

that algorithms based only upon random samples are unstable, making it necessary to use memory. We recall the recent work of Tassiulas [27], which presents a simple randomized algorithm that uses memory for achieving 100% throughput. We present a derandomized version of Tassiulas' algorithm and prove that it is also stable (in Theorem 3). Lemma 1 states a simple criterion for the "goodness" of a switch algorithm, which may be useful elsewhere.

The derandomization mentioned above leads to the algorithm APSARA in Section III-A. APSARA is shown to be stable and simulations show that its delay performance is very competitive compared with MWM. In Section III-B, we present a randomized algorithm called LAURA, which uses memory and outperforms Tassiulas' scheme in terms of delay. It is based on the observation that the weight of a heavy matching is carried in a few of its edges; therefore, it is better to remember heavy edges than it is to remember matchings. Finally, in Section III-C, we propose an algorithm called SERENA, which uses the randomness in the arrivals process for finding good matchings to provide very low delays.

In Section IV, we consider the stability property of these algorithms under general admissible traffic using fluid-model technique. We present simulations to show that these algorithms perform well even under correlated traffic.

II. Throughput

We first define some notations which will be used in the rest of the paper. A matching matrix $S = [S_{ij}]$ can be represented equivalently as a permutation π via the equation $\pi(i) = j$ iff $S_{ij} = 1$ (i.e., if input i is connected to output j under matching S, then i is mapped j under permutation π). Thus, the matching

$$S = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

is equivalent to the permutation $(\pi(1), \pi(2), \pi(3)) = (2, 1, 3)$. Let $Q_{ij}(t)$ denote the queue length of VOQ_{ij} at time t. The weight of matching $S(t)$ is defined as: $\langle S, Q(t)\rangle = \sum_{i,j} S_{ij} Q_{ij}(t)$. Given the queue lengths at time t, $S^\star(t)$ is used to denote the corresponding maximum weight matching and $W^\star(t) = \langle S^\star(t), Q(t)\rangle$ to denote its weight.

As mentioned in the introduction, randomized algorithms are particularly simple to implement because they work on a few randomly chosen samples rather than on the whole state space. As a simple randomized approximation to MWM, consider the following algorithm.

A. Algo1

The MWM algorithm finds, from amongst the $N!$ possible matchings, that matching whose weight is the highest. An obvious randomization of MWM yields the following algorithm, Algo1: At each time t, let the schedule $S(t)$ used by Algo1 be the heaviest of d $(d > 1)$ matchings chosen uniformly at random.

The following theorem shows that Algo1 is not stable, even when $d = O(N)$.

Theorem 1: For an $N \times N$ switch and for any $d \leq cN$, where $c > 0$, Algo1 does not deliver 100% throughput.

Proof: Consider the edge E_{ij} between input i and output j. This edge is present in the schedule, $S(t)$, at time t, only if it belongs to at least one of the d randomly chosen matchings. Consider

$$\begin{aligned} p_{ij} &= P(E_{ij} \in \text{one of the } d \text{ random matchings}) \\ &= 1 - P(E_{ij} \notin \text{any of the } d \text{ random matchings}) \\ &= 1 - P(E_{ij} \notin \text{one random matching})^d \\ &= 1 - \left(1 - \frac{1}{N}\right)^d \\ &\leq 1 - \left(1 - \frac{1}{N}\right)^{cN} \qquad \text{for } d \leq cN \\ &\to 1 - e^{-c}. \end{aligned}$$

Therefore, the service rate available for packets from input i to output j is at most $1 - e^{-c} < 1$. And, as soon as $\lambda_{ij} > 1 - e^{-c}$, we have that the switch is unstable under Algo1. □

Remark: Note that the above theorem has a much stronger implication: *Any* scheduling algorithm that only uses $d = O(N)$ random matchings cannot achieve 100% throughput. Further, there is no assumption about the distribution of the packet arrival process, only a rate assumption. This adds strength to the next algorithm, Algo2, due to Tassiulas [27].

B. Algo2: A Randomized Scheme With Memory

Consider the following algorithm, Algo2.

a) Let $S(t)$ be the schedule used at time t.
b) At time $t+1$ choose a matching $R(t+1)$ uniformly at random from the set of all $N!$ possible matchings.
c) Let $S(t+1) = \arg \max_{S \in \{S(t), R(t+1)\}} \langle S, Q(t+1)\rangle$.

Theorem 2 (Tassiulas [27]): Algo2 is stable under any Bernoulli i.i.d. admissible input.

C. Algo3: A Derandomization of Algo2

Before presenting the algorithm we need the concept of a Hamiltonian walk on the set of all matchings. Consider a graph with $N!$ nodes, each corresponding to a distinct matching, and all possible edges between these nodes. Let $Z(t)$ denote a Hamiltonian walk on this graph; that is, $Z(t)$ visits each of the $N!$ distinct nodes exactly once during times $t = 0, \ldots, N! - 1$. We extend $Z(t)$ for $t \geq N!$ by defining $Z(t) = Z(t \bmod N!)$. One simple algorithm for such a Hamiltonian walk is described, for example, in [22, Ch. 7]. This is a very simple algorithm that requires $O(1)$ space and $O(1)$ time, to generate $Z(t+1)$ given $Z(t)$. Under this algorithm $Z(t)$ and $Z(t+1)$ differ in exactly two edges. For $N = 3$ this algorithm generates the matchings: $Z(0) = (1,2,3)$, $Z(1) = (1,3,2)$, $Z(2) = (3,1,2)$, $Z(3) = (3,2,1)$, $Z(4) = (2,3,1)$, $Z(5) = (2,1,3)$, $Z(6) = Z(0)$, and $Z(7) = Z(1), \ldots$.

Now consider Algo3.

a) Let $S(t)$ be the schedule used at time t.
b) At time $t+1$ let $R(t+1) = Z(t+1)$, the matching visited by the Hamiltonian walk.
c) Let $S(t+1) = \arg \max_{S \in \{S(t), R(t+1)\}} \langle S, Q(t+1)\rangle$.

We shall prove the stability of ALGO3 after establishing the following lemma.

Lemma 1: Consider an input-queued switch with admissible Bernoulli i.i.d. inputs. Let $Q(t)$ be the queue-size process that results when the switch uses scheduling algorithm B. Let $W^B(t)$ denote the weight of the schedule used by B at time t, and let $W^\star(t)$ be the weight of MWM given the same queue-size process $Q(t)$. If there exists a positive constant c such that the property

$$W^B(t) \geq W^\star(t) - c$$

holds for all t, then the algorithm B is stable.

Proof: To establish stability it suffices to prove that (for example, see [15] and [16]) for some $\delta > 0$ and $K > 0$

$$\begin{aligned} E\left(V\left(Q(t+1)\right) - V\left(Q(t)\right)|Q(t)\right)& \\ \leq -\delta W^\star(t), &\quad \text{whenever } W^\star(t) \geq K\end{aligned}$$

where $V(Q(t)) = \sum_{i,j} Q_{ij}^2(t)$.

Consider the following:

$$\begin{aligned} V\left(Q(t+1)\right) - V\left(Q(t)\right) &= \sum_{i,j} \left[Q_{ij}^2(t+1) - Q_{ij}^2(t)\right] \\ &= \sum_{i,j} \left[Q_{ij}(t+1) - Q_{ij}(t)\right] \\ &\quad \times \left[Q_{ij}(t+1) + Q_{ij}(t)\right].\end{aligned}$$

Let $S(t)$ be the schedule used by B at time t and let $A_{ij}(t)$ denote arrivals to VOQ_{ij} at time t. We know that

$$\begin{aligned} Q_{ij}(t+1) &= [Q_{ij}(t) - S_{ij}(t)]^+ + A_{ij}(t+1) \\ &\leq \max\left\{[Q_{ij}(t) - S_{ij}(t)] + A_{ij}(t+1), 1\right\}.\end{aligned}$$

Hence, we obtain

$$\begin{aligned} V\left(Q(t+1)\right) - V\left(Q(t)\right) &\leq \sum_{i,j} [(A_{i,j}(t+1) - S_{ij}(t)) \\ &\quad \times (2Q_{ij}(t) + 1) + 1] \\ &\leq \sum_{i,j} [(A_{i,j}(t+1) - S_{ij}(t)) \\ &\quad \times (2Q_{ij}(t))] + 2N^2.\end{aligned}$$

Taking conditional expectations with respect to $Q(t)$ yields

$$\begin{aligned} E\left(V\left(Q(t+1)\right) - V\left(Q(t)\right)|Q(t)\right) &\leq 2\sum_{ij} Q_{ij}(t)\left[E\left(A_{ij}(t)\right.\right. \\ &\quad \left.\left. -\ S_{ij}(t)|Q(t)\right)\right] + 2N^2 \\ &= 2\sum_{ij} Q_{ij}(t)\left[\lambda_{ij}\right. \\ &\quad \left. -\ S_{ij}(t)\right] + 2N^2.\end{aligned}$$

Since the arrival rate matrix, Λ, is admissible it is strictly doubly substochastic. Therefore, from arguments made in [16, Lemma 2], we may write $\sum_{ij} Q_{ij}(t)\lambda_{ij} = \langle Q(t), \Lambda\rangle \leq \sum_k \gamma_k \langle \Pi_k, Q(t)\rangle$, where the Π_k are permutation matrices and $\gamma_k \geq 0$ and $\sum_k \gamma_k < 1$.

Let $W_{\Pi_k} = \langle \Pi_k, Q(t)\rangle$ and let $\delta = 1 - \sum_k \gamma_k$. Putting the above observations together, we get

$$\begin{aligned} &E\left(V\left(Q(t+1)\right) - V\left(Q(t)\right)|Q(t)\right) \\ &\leq 2\left(\sum_k \gamma_k W_{\Pi_k}(t) - W^B(t)\right) + 2N^2 \\ &= 2\left(\sum_k \gamma_k W_{\Pi_k}(t) - W^\star(t) + W^\star(t) - W^B(t)\right) + 2N^2 \\ &\leq 2\left(\sum_k \gamma_k - 1\right) W^\star(t) + 2c + 2N^2 \\ &= -2\delta W^\star(t) + C, \text{ where } C = 2c + 2N^2.\end{aligned}$$

Hence, for large enough constant $K > 0$, we obtain for $W^\star(t) \geq K$

$$E\left(V\left(Q(t+1)\right) - V\left(Q(t)\right)|Q(t)\right) \leq -\delta W^\star(t).$$

This proves the stability of algorithm B. □

Theorem 3: An input-queued switch using ALGO3 is stable under all admissible Bernoulli i.i.d. inputs.

Proof: Since there is at most one packet arriving at or departure from each VOQ in each time slot, we obtain for any matching M that

$$\langle M, Q(t)\rangle \geq \langle M, Q(t+s)\rangle - sN. \tag{1}$$

Let $S(t)$ denote the schedule used by ALGO3 at time t, and let $W^{(3)}(t) = \langle S(t), Q(t)\rangle$ be its weight. If, for every time t, it holds that $W^{(3)}(t) \geq W^\star(t) - c$ for some $c > 0$, then by Lemma 1 it follows that ALGO3 is stable.

Consider a specific time instant T. Let S_1 and S_0 denote the maximum weight matchings at time T and $T - N!$, respectively. Now, by the property of the Hamiltonian walk, there is a $t' \in [T - N!, T]$ such that $Z(t') = S_0$. Then

$$\begin{aligned} \langle S(t'), Q(t')\rangle &\overset{(a)}{\geq} \langle S_0, Q(t')\rangle \\ &\overset{(b)}{\geq} \langle S_0, Q(T - N!)\rangle \\ &\quad - (t' + N! - T)N \end{aligned} \tag{2}$$

where (a) follows from the definition of ALGO3 and (b) follows from (1).

For every t, it follows from (1) and the definition of ALGO3 that

$$\langle S(t), Q(t)\rangle - N \leq \langle S(t), Q(t+1)\rangle \leq \langle S(t+1), Q(t+1)\rangle.$$

Using this repeatedly in the following, we obtain:

$$\begin{aligned} \langle S(T), Q(T)\rangle &\geq \langle S(t'), Q(t')\rangle - (T - t')N \\ &\overset{(c)}{\geq} \langle S_0, Q(T - N!)\rangle - NN! \\ &\overset{(d)}{\geq} \langle S_1, Q(T - N!)\rangle - NN! \\ &\overset{(e)}{\geq} \langle S_1, Q(T)\rangle - 2NN! \end{aligned}$$

where (c) follows from (2), (d) follows from the fact that S_0 is the maximum weight schedule at time $(T - N!)$, and (e) follows from (1).

Since T was arbitrary, we have shown that $W^{(3)}(t) \geq W^{\star}(t) - 2NN!$ for every t. This completes the proof of Theorem 3. □

Lemma 1 and Theorem 3 together provide a general method for establishing the stability of algorithms whose weight is "good enough." Thus, they may be applicable to a wider class of algorithms than those that use memory.

III. Delay

For a scheduling algorithm to have a good delay performance in addition to providing 100% throughput, it needs to do extra work. In the following sections, we describe three different algorithms that respectively use parallelism, randomization and the information in arrivals to achieve 100% throughput *and* a good delay performance.

A. APSARA

As noted in the introduction, determining the maximum weight matching essentially involves a search procedure, which can take many iterations and be time-consuming. Since our goal is to design high-performance schedulers for high-aggregate bandwidth switches, algorithms that involve too many iterations are unattractive.

Our goal is to design a high-performance scheduler that only requires a *single* iteration. Therefore, we must devise a fast method for finding good schedules. One method for speeding up the scheduling process is to search the space matchings in parallel. Fortunately, the space of matchings has a nice combinatorial structure which can be exploited for conducting efficient searches. In particular, it is possible to query the "neighbors" of the current matching in parallel and use the heaviest of these as the matching for the next time slot. This observation inspires the APSARA algorithm, which employs the following two ideas:

1) use of memory;
2) exploring neighbors in parallel. The neighbors are defined such that it is easy to compute them using hardware parallelism.

Definition 1: (Neighbor) Given a permutation π, let S be the corresponding matching: $S_{i\pi(i)} = 1$ for all i. A matching S' is said to be a neighbor of S iff there are exactly two inputs, say i_1 and i_2, such that S' connects input i_1 to output $\pi(i_2)$ and input i_2 to output $\pi(i_1)$. All other input–output pairs are the same under S and S'. The set of all neighbors of a matching S is denoted $\mathcal{N}(S)$.

Essentially, a neighbor, S', of S is obtained by swapping two edges in S, leaving the other $N-2$ edges of S fixed. Note that the cardinality of $\mathcal{N}(S)$ is $\binom{N}{2}$. For example, the matching S for a 3×3 switch and its three neighbors S_1, S_2, and S_3 are given below

$$S = (1,2,3) \quad S_1 = (2,1,3) \quad S_2 = (1,3,2) \quad S_3 = (3,2,1).$$

1) APSARA: The Basic Version: Let $S(t)$ be the matching determined by APSARA at time t. Let $Z(t+1)$ the matching corresponding to the Hamiltonian walk at time $t+1$. At time $t+1$ APSARA does the following.

1) Determine $\mathcal{N}(S(t))$ and $Z(t+1)$.
2) Let $\mathcal{M}(t+1) = \mathcal{N}(S(t)) \cup Z(t+1) \cup S(t)$. Compute the weight $\langle S', Q(t+1)\rangle$ for all $S' \in \mathcal{M}(t+1)$.
3) The matching at time $t+1$ is given by
$$S(t+1) = \arg \max_{S' \in \mathcal{M}(t+1)} \langle S', Q(t+1)\rangle .$$

APSARA requires the computation of the weight of neighbor matchings. Each such computation is easy to implement since a neighbor S' differs from the matching $S(t)$ in exactly two edges. However, computing the weights of all $\binom{N}{2}$ neighbors requires a lot of space in hardware for large values of N.

To overcome this, we make a different definition of what it means to be a neighbor, thereby restricting the size of the neighborhood set. In particular, we are aiming for a neighborhood of size $O(N)$, as opposed to the order $O(N^2)$ as in APSARA.

Definition 2: (Linear Neighbor) A matching S' is said to be a linear neighbor of another matching S iff there are exactly two inputs, i_1 and $i_2 \triangleq (i_1 + 1 \bmod N)$, such that S' connects input i_1 to output $\pi(i_2)$ and input i_2 to output $\pi(i_1)$. All other input–output pairs are the same under S and S'. The set of all neighbors of a matching S is denoted $\mathcal{N}_L(S)$.

Note that the cardinality of $\mathcal{N}_L(S)$ is exactly N. Denote by APSARA-L the version of the basic APSARA algorithm when neighbors are chosen from $\mathcal{N}_L(S)$.

Further, suppose that hardware space constraints allow the use of at most $K \ll N$ modules, then how can the search procedure required by APSARA(or APSARA-L) be conducted efficiently?

One obvious solution is to search the neighborhood set over multiple iterations by reusing the K modules. After all, at low line speeds there is more time for scheduling packets, allowing one to conduct more iterations. However, if line speeds are high and one is only allowed *one iteration*, then the question arises as to which K neighbors should be chosen. A deterministic procedure for choosing the K neighbors will usually result in poor choices since, a priori, it is not clear which neighbors are heavy. It is better to choose K neighbors *at random* and use the heaviest of these. This motivates the following variant of APSARA.

2) APSARA-R: The Randomized Variant: Suppose hardware constraints only allow us to query K neighbors. Let $\mathcal{N}_K(S(t))$ denote the set of K elements picked uniformly at random from the set $\mathcal{N}(S(t))$. APSARA-R determines the matching $S(t+1)$ as follows.

1) Determine $\mathcal{N}_K(S(t))$ (note that it is not necessary to generate $\mathcal{N}(S(t))$). Determine $Z(t+1)$, the status of the Hamiltonian walk.
2) Let $\mathcal{M}_K(t+1) = \mathcal{N}_K(S(t)) \cup Z(t+1) \cup S(t)$. Compute $\langle S', Q(t+1)\rangle$ for every $S' \in \mathcal{M}_K(t+1)$.
3) $S(t+1) = \arg \max_{S' \in \mathcal{M}_K(t+1)} \{W_{S'}(t+1))\}$.

Remark: We conclude the description of APSARA by mentioning one last point. APSARA generates all the matchings in the neighborhood set oblivious of the current queue lengths. The queue lengths are only used to select the heaviest matching from the neighborhood set. It is, therefore, possible that the matching determined by APSARA, while being heavy, is not of maximal size. That is, there exists an input, say i, which has packets for an output j, but the matching $S(t)$ connects input i to some other output j' and connects output j some other input i', and both

$Q_{ij'}(t)$ and $Q_{i'j}(t)$ are equal to zero. Thus, input i and output j will both idle unnecessarily.

If needed, it is easy to complete the matching $S(t)$ determined by APSARA into a maximal matching. We shall call the maximal version MaxAPSARA. There are several simple ways to maximize APSARA, and pretty much any one can be chosen. We note from simulations that the maximization step leads to relatively very small improvements in the performance of APSARA and, therefore, may be avoided altogether.

3) APSARA Theorems:

Theorem 4: The algorithms APSARA, APSARA-L, and APSARA-R are all stable under admissible Bernoulli i.i.d. inputs.

Proof: All versions use the Hamiltonian walk. Therefore, Lemma 1 and Theorem 3 apply and the stability of algorithms follows. □

Theorem 5: Let $S(s)$ denote the schedule obtained by APSARA at time s, and let $W^S(s) = \langle S(s), Q(s)\rangle$ denote its weight. If $S(t) = S(t-1)$, that is the schedule does not change from time $t-1$ to time t, then

$$W^S(t) \geq \frac{1}{2} W^\star(t)$$

where $W^\star(t)$ is the weight of maximum weight matching at time t.

Proof: Without loss of generality, assume that the maximum weight matching, $S^\star(t)$, at time t is the identity permutation; that is, input i is matched to output i under the maximum weight matching. Let the permutation corresponding to the schedule $S(t)$ be π. That is, $S(t)$ matches input i to output $\pi(i)$. Let w_{ij} denote the weight of VOQ_{ij} at time t. Consider any $i, 1 \leq i \leq N$. Suppose $\pi(i) \neq i$. Let $\pi^{-1}(i)$ be the input matched to output i under $S(t)$. Since $S(t-1) = S(t)$, from the property of APSARA, it follows that for every i

$$w_{i\pi(i)} + w_{\pi^{-1}(i)i} \geq w_{ii}.$$

Now, summing over i, we obtain

$$\sum_i w_{i\pi(i)} + w_{\pi^{-1}(i)i} \geq \sum_i w_{ii}.$$

But, $\sum_i w_{i\pi(i)} = \sum_i w_{\pi^{-1}(i)i}$, since π is a permutation and, hence

$$\sum_i w_{i\pi(i)} \geq \frac{1}{2} \sum_i w_{ii}.$$

Now $\sum_i w_{i\pi(i)}$ is the weight of the APSARA schedule and $\sum_i w_{ii}$ is the weight of the maximum weight matching. Thus, $W^S(t) \geq (1/2) W^\star(t)$ and the theorem is proved. □

4) Implementation: All versions of APSARA involve a Hamiltonian walk. This was done for purely theoretical reasons: to ensure their stability (Theorem 4). We have found that, in practice, the Hamiltonian walk is not necessary; that is, the algorithms provide virtually the same delay and throughput even without it. Thus, while the walk is extremely simple to implement, we do not consider it either in implementation or in performance evaluation.[3]

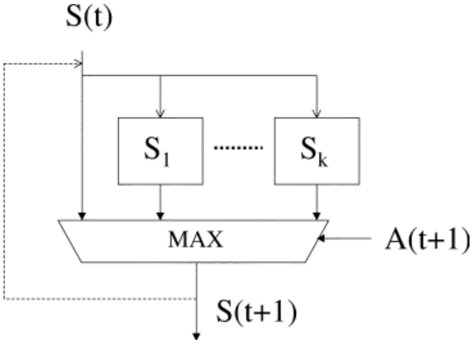

Fig. 3. Schematic for the implementation of APSARA. The old matching $S(t)$ and the new arrivals $A(t+1)$, are used to compute the weights of the k neighbor matchings in parallel. The new matching $S(t+1)$ is the one with highest weight among all the neighbors. Note that this architecture is parallel and can be easily pipelined.

The main feature of APSARA is that it can be implemented in a parallel architecture very efficiently. Fig. 3 shows a schematic for the implementation of APSARA with K modules.

5) The Simulation Setting: Before presenting the performance of APSARA, we outline the simulation setting that will be used throughout the rest of the paper. We have conducted extensive simulations of all the algorithms we present under all the different types of traffic mentioned below. In addition, we have also conducted simulations of switches with 64 and 1024 ports. Due to limitations of space and for uniformity of comparison, we only present a subset of simulations which represent "critical" loading conditions. Fig. 13 shows the average queue length of each VOQ for different algorithms under uniform traffic. Not surprisingly, all algorithms perform well under this loading uniform traffic; thus, it is not "critical." More extensive simulations may be found in [9] and [25].

Switch: number of ports: $N = 32$. Each VOQ can store up to 10 000 packets. Excess packets are dropped.

Input Traffic: All inputs are equally loaded on a normalized scale, and $\rho \in (0, 1)$ denotes the normalized load. The arrival process is Bernoulli i.i.d.

Let $|k| = (k \bmod N)$. The following load matrices are used to test the performance of APSARA.

1) *Uniform:* $\lambda_{ij} = \rho/N \, \forall i, j$. This traffic does not test much since all algorithms perform well under it (see Fig. 13).
2) *Diagonal:* $\lambda_{ii} = 2\rho/3$, $\lambda_{i|i+1|} = \rho/3 \, \forall i$, and $\lambda_{ij} = 0$ for all other i and j. This is a very skewed loading, in the sense that input i has packets only for outputs i and $|i+1|$. It is more difficult to schedule than uniform loading.
3) *Logdiagonal:* $\lambda_{ij} = 2\lambda_{i|j+1|}$ and $\sum_i \lambda_{ij} = \rho$. For example, the distribution of the load at input 1 across outputs is: $\lambda_{1j} = 2^{N-j}\rho/(2^N - 1)$. This type of load is more balanced than diagonal loading, but clearly more skewed than uniform loading. Hence, the performance of a specific algorithm becomes worse as we change the loading from uniform to logdiagonal to diagonal. In this paper, we do not presents simulation results for logdiagonal traffic, since they are qualitatively similar to the results for diagonal traffic.

Performance Measures: We compare the queue lengths induced by different algorithms, the delays can be computed using

[3] Note that eliminating the Hamiltonian walk can only worsen the performance, the actual algorithms perform even better.

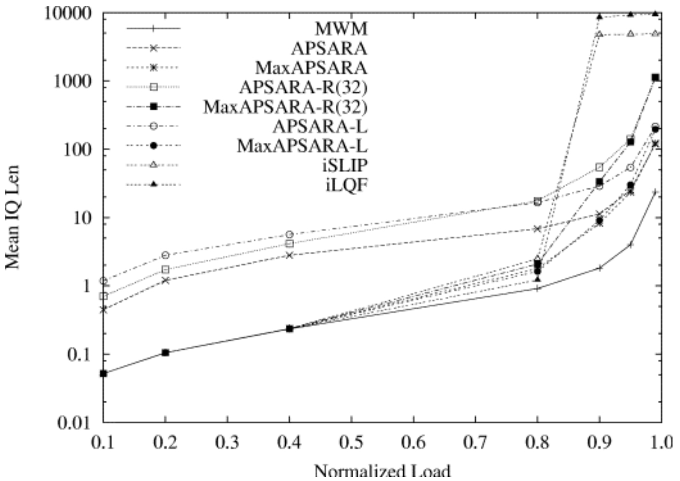

Fig. 4. Mean IQ length for APSARA under diagonal traffic.

Little's Law.[4] The simulations are run until the estimate of the average delay reaches the relative width of the confidence interval equal to 1% with probability $\geq 95\%$. The estimation of the confidence interval width uses the *batch means* approach.

Fig. 4 compares the average queue sizes induced by APSARA, MWM, iSLIP (with N iterations), and iLQF (with N iterations) under diagonal traffic. As seen, APSARA and MaxAPSARA perform very competitively with MWM under all loadings. On the other hand, both iLQF and iSLIP incur severe packet losses and delays under heavy loading. We also note that under low loads, APSARA deviates from MaxAPSARA since it is not maximal. Therefore, it may cause certain VOQs to idle. But, the difference is very small—no more than ten packets on average.

We see that APSARA-L only 32 modules, performs quite well when compared with APSARA, which uses $\binom{32}{2} = 496$ modules; even at high loads, the difference between queue sizes is very small. While APSARA-R(32) does not perform, as well as APSARA-L, when the number of modules $K \ll N$, then randomization appears to be the best option.

B. LAURA

As shown by Tassiulas [27], ALGO2 provides 100% throughput. However, its delay performance is quite poor (as we will see in Fig. 6). This is because of its particular use of memory: it carries matchings between iterations via memory. But, when the weight of a heavy matching resides in a few heavy edges, it is more important to remember the heavy edges than it is to remember the matching itself. This simple observation motivates the next algorithm LAURA, which iteratively augments the weight of the current matching by combining its heavy edges with the heavy edges of a (nonuniformly) randomly chosen matching.

There are three main features in the design of LAURA:

1) use of memory;
2) nonuniform random sampling;
3) a merging procedure for weight augmentation.

1) The LAURA Algorithm: Let $S(t)$ be the matching used by LAURA at time t. At time $t+1$ LAURA does the following:

a) generate a random matching $R(t+1)$ using the RANDOM procedure.
b) use $S(t+1) = \text{Merge}(R(t+1), S(t))$ as the schedule for time $t+1$.

Random Procedure: Let $\mathcal{F}_\eta(M)$ denote the minimal set of edges in the matching M carrying at least a fraction η $(0 \leq \eta \leq 1)$ of its weight. We shall call η the *selection factor*.

R*ANDOM* is the following iterative procedure: Initially, all inputs and outputs are marked as *unmatched*. The following steps are repeated in each of I iterations, where I is typically $\log_2 N$.

1) Let i be the current iteration number. Let $k \leq N$ be the number of *unmatched* input–output pairs. Out of the $k!$ possible matchings between these unmatched input–output pairs, a matching $S_i(k)$ is chosen uniformly at random.
2) If $i < I$, retain the edges corresponding to $\mathcal{F}_\eta(S_i(k))$ and mark the nodes they cover as *matched*. If $i = I$, then retain all edges of $S_i(k)$.

Merge Procedure: Given a bipartite graph and two matchings $M1$ and $M2$ for this graph, the MERGE procedure returns a matching $\tilde{M}$ whose edges belong either to $M1$ or to $M2$. MERGE works as follows.

Color the edges of $M1$ red and the edges of $M2$ green. Start at output node j_1 and follow the red edge to an input node, say i_1. From input node i_1 follow the (only) green edge to its output node, say j_2. If $j_2 = j_1$, stop. Else continue to trace a path of alternating red and green edges until j_1 is visited again. This gives a "cycle" in the subgraph of red and green edges.

Suppose the above cycle does not cover all the red and green edges. Then, there exists an output j outside this cycle. Starting from j repeat the above procedure to find another cycle. In this fashion, find all cycles of red and green edges. Suppose there are m cycles, $C_1, \ldots, C_m$ at the end. Then, each cycle C_i contains two matchings: G_i which has only green edges, and R_i which has only red edges. The MERGE procedure returns the matching

$$\tilde{M} = \bigcup_{i=1}^{m} \arg \max_{S \in \{G_i, R_i\}} \langle S, Q(t) \rangle .$$

Fig. 5 illustrates the MERGE procedure. It is easy to show that the final matching $\tilde{M}$ is the maximum weight matching on the subgraph defined by edges of $M1$ and $M2$.

2) LAURA: Complexity and Stability: It can be shown that the running time of LAURA is bounded by $O(IN \log_2 N + N)$. In our simulation study, we set $I = \log_2 N$. Thus running time of algorithm is $O(N \log^2 N)$.

The following theorem is about the stability of LAURA.

Theorem 6: LAURA is a stable algorithm, i.e., it achieves 100% throughput under admissible Bernoulli i.i.d. inputs.

Proof: This follows from the proof of Theorem 2, since the probability that $R(t+1)$ equals the maximum weight matching is lower bounded by a positive constant for all time. And, as shown in Theorem 2, this is sufficient to ensure its stability. □

[4] Note that Little's Law holds also for nonwork-conserving stable systems, like IQ switches.

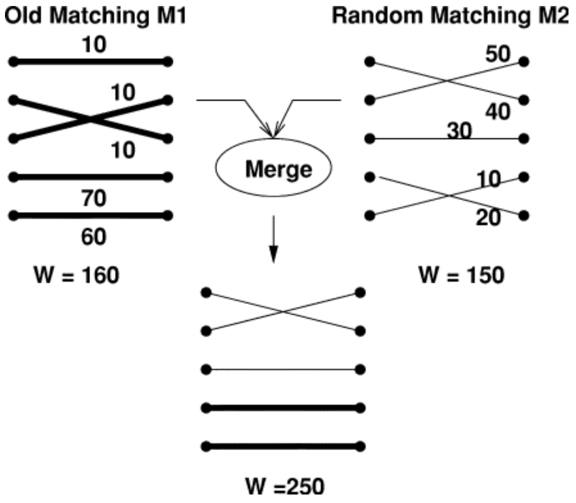

Fig. 5. An illustration of the MERGE applied to matchings M1 and M2. The final matching is the maximum weight matching on the subgraph defined by edges of M1 and M2.

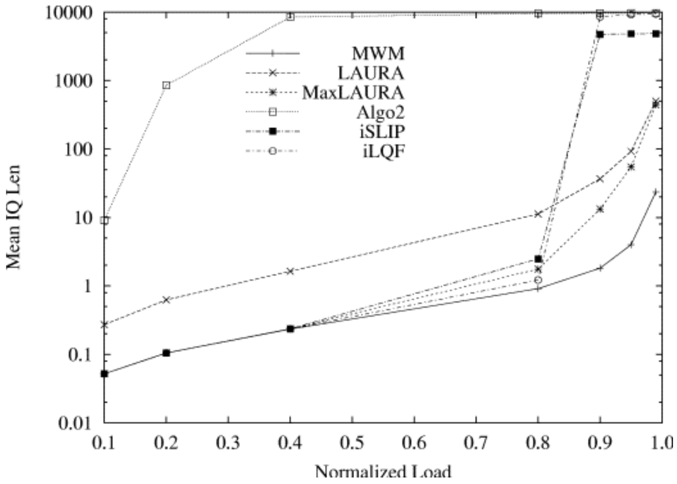

Fig. 6. Mean IQ length for LAURA under diagonal traffic.

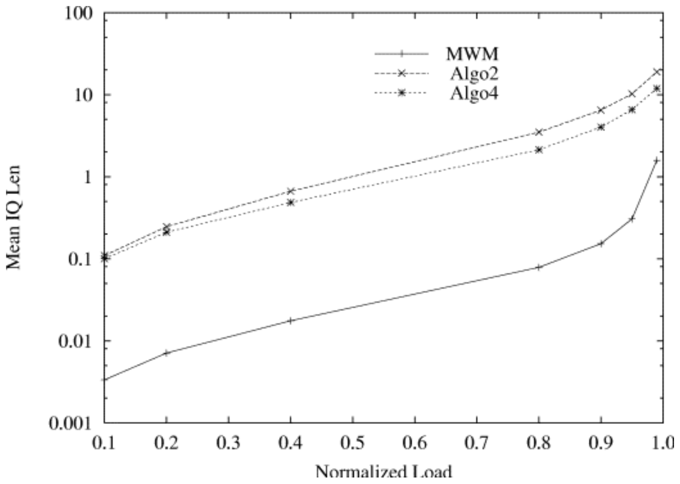

Fig. 7. Mean IQ length under uniform traffic for ALGO2 and ALGO4. The two algorithms behave almost the same.

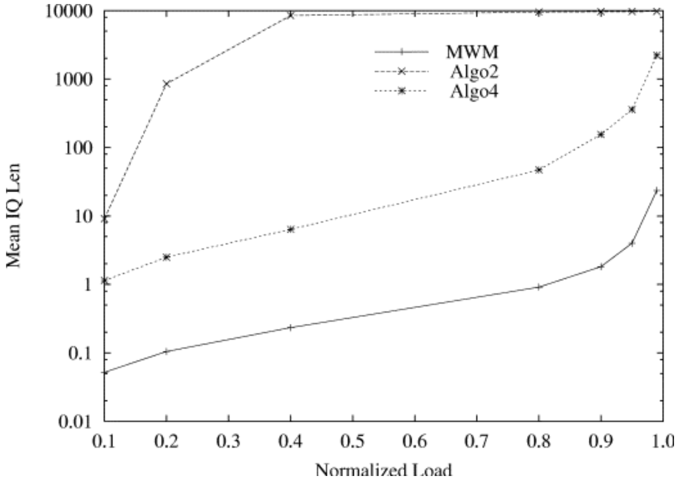

Fig. 8. Mean IQ length under diagonal traffic for ALGO2 and ALGO4. ALGO4 shows a much better behavior than ALGO2, illustrating the goodness of the MERGE procedure.

3) Performance: The simulation setting is identical to that for the APSARA algorithm. We set the selection factor $\eta = 0.5$, and the number of iterations $I = 5 = \log_2 N$. LAURA is compared with the MWM, iSLIP, iLQF, and ALGO2 algorithms under diagonal traffic. The results are shown in Fig. 6. The algorithms LAURA and MaxLAURA (which outputs a maximal matching, similarly to what happens with MaxAPSARA) perform quite competitively with respect to MWM. We see that iSLIP and iLQF suffer large packet losses at high loads. Strangely enough, although ALGO2 is provably stable (as opposed to iSLIP and iLQF), its performance in terms of average backlog is the worst. Note that this is not surprising, if the Lyapunov's criteria for the stability is carefully understood. The switching system is stable if infinite queue sizes are allowed. This fact, in some sense, gives stronger motivation for the algorithms we propose in this paper, since they achieve 100% throughput (like ALGO2) but with delays very low and comparable with the MWM algorithm.

4) Role of the Merge Procedure: In this section, we study the role of the MERGE procedure in LAURA for obtaining good delay performance.

We consider the following two simple algorithms: ALGO2 (by Tassiulas) and its variant called ALGO4, in which: $S(t+1) = \text{Merge}(R(t+1), S(t))$.

Figs. 7 and 8 show the average queue lengths for these two algorithms. Fig. 7 shows that both algorithms behave almost the same under uniform traffic and, thus, the MERGE procedure does not make a big difference to the performance under this traffic. When the traffic is not uniform, as shown in Fig. 8, ALGO4 performs much better compared with ALGO2. This shows that the use of the MERGE procedure is essential for obtaining good delay performance under nonuniform traffic.

5) Learning Time: MERGE Versus MAX: The main reason behind achieving 100% throughput for algorithms like ALGO2 and ALGO4 is the finite amount of time (on average) that it takes these algorithms to obtain a matching whose weight is comparable to that of MWM. But the learning time can be drastically different and this directly affects the delay performance of the underlying scheduling algorithm.

We now make a comparison of the learning time of ALGO4, which uses MERGE procedure, with that ALGO2, which uses MAX procedure. First, we present the simulation study under

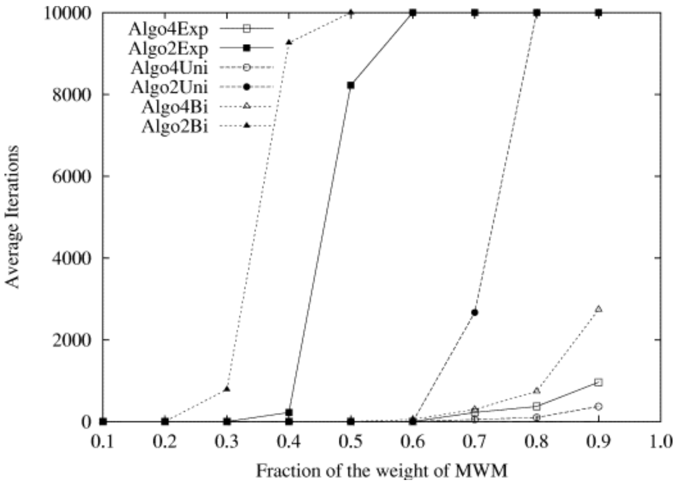

Fig. 9. The comparison of learnning time between ALGO2 and ALGO4 under different weight distributions: uniform ("Uni"), exponential ("Exp"), and bimodal ("Bi"). Only ALGO4 adopting MERGE procedure is able to learn 90% of the weight of the MWM in a small number of iterations.

different scenarios and then present analytical results to understand the observed behavior under a simple model.

Simulation setting: A random weighted bipartite graph is created by choosing the weight of each edge according to independent and identically distributed random variables with mean 1. We consider three different distributions: 1) exponential; 2) uniform on [0, 2]; and 3) bimodal on $\{0.1, 9.1\}$ with probabilities $\{0.9, 0.1\}$.

Both algorithms ALGO2 and ALGO4 start with same random initial matching and subsequently they are provided with the same random matchings. Both the algorithms run till they obtain a matching whose weight is at least a pre-determined fraction f of the weight of MWM on the same graph. The average number of iterations taken by an algorithm to achieve this weight is used as a measure of its learning time. When an algorithm takes more than 10 000 iterations to learn this weight, we simply report the number of iterations as 10 000.

For each $f \in [0.1, 0.9]$, and for each distributions, we obtain the average number of iterations over 100 sample runs. The results are plotted in the Fig. 9. It shows that for all distributions, both algorithms manage to learn quickly when $f \leq 0.2$. But as f grows the average number of iterations taken by ALGO2 is very high compared to that of ALGO4. We also note that learning time gets worse as the variance of the edge-weight distribution increases; i.e., uniform is easier to learn than exponential distribution which is easier to learn than bimodal distribution.

MERGE Versus MAX: Analytical Results: The simulation study showed that ALGO4 learns "good" matchings a lot quicker compared with ALGO2 under different edge-weight distributions. It is not as easy to obtain such qualitative results analytically for any general edge-weight distribution. As our interest is in determining the learning time of an algorithm, we consider a simplified model in which the edges of the maximum weight matching are assigned weight ∞ (or a large enough vaule) and all other edges are assigned weight 0. We will then be interested in theoretically understanding the average time taken to learn the MWM.

Without loss of generality, assume that the MWM is the identity matching: i.e., the edge-weight matrix of the bipartite graph has ∞ on the N entries of the main diagonal and zero on the remaining $N^2 - N$ positions. We compare the performance of ALGO2 and ALGO4 in this context. Each time both algorithms are provided the same random matching. The MWM is learned when all edges of the identity matching are learned by the algorithm.

First, consider the performance of ALGO2. Note that the matching retained by ALGO2 at the end of iteration t will be the matching with the most of edges in common with the identity matching, among all random matchings chosen till iteration t.

An edge i of a matching is said to be *fixed* if it matches input i to output i. Note that all the elements of the identity matching are fixed. To understand the learning time of an algorithm, it is therefore useful to study the distribution of the number of fixed edges in a randomly chosen permutation. This distribution is well-studied in the literature in various contexts.

Let A_i denote the event that i^{th} element is fixed in a randomly chosen permutation R. Let P_k^N denote the probability that exactly k elements are fixed in a randomly chosen permutation of size N. First, let us compute P_0^N: the probability that no element is fixed

$$\begin{aligned} P_0^N &= \Pr\left(\cap_{i=1}^N A_i^c\right) = 1 - \Pr\left(\cup_{i=1}^N A_i\right) \\ &\overset{(a)}{=} 1 - \sum_{j=1}^{N} (-1)^{j+1} \binom{N}{j} \frac{(N-j)!}{N!} \\ &= \sum_{j=1}^{N} (-1)^j \frac{1}{j!} \approx e^{-1} \end{aligned}$$

where (a) is direct application of inclusion-exclusion principle. Now for any k

$$P_k^N = \frac{\binom{N}{k} P_0^{N-k} (N-k)!}{N!} = O\left(\frac{1}{k!}\right)$$

and

$$Q_k^N = \sum_{j \geq k} P_j^N = O\left(\sum_{j \geq k} \frac{1}{j!}\right) = O\left(\frac{1}{k!}\right) \tag{3}$$

is the probability that there are at least k fixed elements in a randomly chosen matching. Thus, on average, ALGO2 takes $(1/(Q_k^N)) = O(k!)$ iterations to learn k fixed elements or k elements of MWM.

We will now show that the order of the learning time for ALGO4 is significantly smaller than that of ALGO2. Let $M(t)$ denote the matching retained by ALGO4 at the end of iteration t, and $R(t+1)$ be the random matching chosen at iteration $t+1$. Then, $M(t+1)$ results by $M(t)$ with $R(t+1)$.

Now, the bipartite graph containing the edges of $M(t)$ and $R(t+1)$ is made up of cycles with edges alternatively belonging to $M(t)$ and $R(t+1)$. This is the same as the cyclic decomposition of a random a permutation. In each cycle, the MERGE procedure either picks all edges from $M(t)$ or all edges from $R(t+1)$. Hence, it is important to know the distribution of cycles in a random permutation. We briefly recall the salient features of this well-studied distribution. Let $K(t)$ be the random variable representing the number of cycles in the graph defined by $M(t)$ and

$R(t+1)$, and let $C_l(t), 1 \le l \le K(t)$ be the length of lth cycle. It is well-known that $K(t)$ is sharpy concentrated around its mean: $\log_e N$. Although the distribution of cycle-lengths $C_l(t)$ is not concentrated around its mean, $N/\log_e N$, for simplicity we assume the following: there are $\log_e N$ cycles each of length $N/\log_e N$. (It can be shown that this assumption gives a weaker upper bound on the learning time of ALGO4.)

Let $X(t)$ be the number of fixed elements in $M(t)$; that is the elements of MWM already learnt by t. We now obtain a lower bound for the probability that the number of fixed elements will increase in $M(t+1)$ by at least one.

Consider the following event: $R(t+1)$ contains a fixed element and it belongs to a cycle which does not contain any of the $X(t)$ fixed elements of $M(t)$. In this case, the ALGO4 will pick elements of $R(t+1)$ for this cycle. This in turn increases the number of fixed elements in $M(t+1)$ to at least $X(t)+1$. We now compute the probability of this event.

The probability that there are k fixed elements in $R(t+1)$ is $O(1/k!)$ as computed above. The $X(t)$ fixed elements of $M(t)$ are distributed among the $\log_e N$ cycles uniformly at random. A cycle contains $N/\log_e N$ elements from each of $R(t+1)$ and $M(t)$. The probability that the cycle containing the fixed element of $R(t+1)$ does not contain any of the $X(t)$ elements is

$$p = \frac{\binom{N-X(t)}{\frac{N}{\log_e N}}}{\binom{N}{\frac{N}{\log_e N}}} \approx \left(1 - \frac{X(t)}{N}\right)^{\frac{N}{\log_e N}}.$$

It follows from the above discussion that

$$\begin{aligned} E\left[X(t+1) - X(t)\right] &\ge \sum_{k\ge 1} \frac{1}{k!} kp = \sum_{k \ge 0} \frac{1}{k!} p \\ &\approx \left(1 - \frac{X(t)}{N}\right)^{\frac{N}{\log N}} \approx \exp\left\{-\frac{X(t)}{\log N}\right\}. \end{aligned}$$

Let $y(s) = E[X(sN)]/N$. Then we obtain the following differential equation for large N:

$$\frac{\partial y(s)}{\partial s} = \exp\left\{-N\frac{y(s)}{\log N}\right\}.$$

The solution of this equation is

$$\frac{\log N}{N}\left(\exp\left\{\frac{Ny(s)}{\log N}\right\} - 1\right) = s \tag{4}$$

which implies

$$\log N \left(\exp\left\{\frac{X(t)}{\log N}\right\} - 1\right) = t. \tag{5}$$

Thus, ALGO4 takes $T_4 = O(\log N \exp\{N/\log N\})$ amount of time to learn MWM, while we have seen earlier that ALGO2 takes $T_2 = O(N!)$ time. These times compare as

$$T_2 = T_4^{\log^2 N}.$$

This shows the drastic difference in the learning times of these two algorithms and the power of the MERGE procedure. While we have made some simplifying assumptions in both the modeling and analysis above, in the future, we plan to investigate graphs with random edge weights (not just zero or ∞) and take into account the details of the cycle-length distribution. This will help to tighten the rather weak bounds derived above, and give a more complete picture of the analysis.

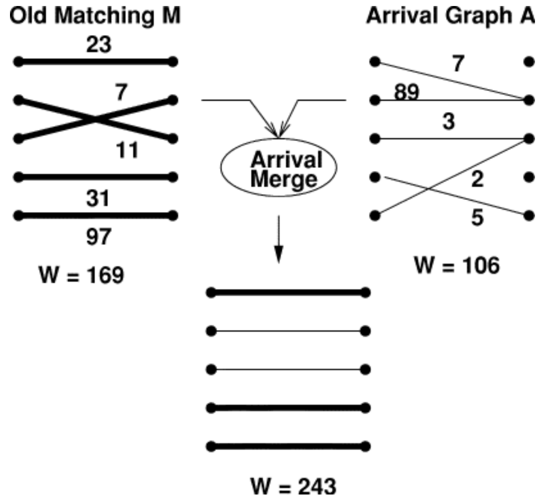

Fig. 10. An illustration of the ARR-MERGE procedure, given the matching M and the arrival graph A.

C. SERENA

Our final algorithm, SERENA is based on the following ideas:

1) use of memory;
2) exploiting the randomness in arrivals;
3) a merging procedure, involving new arrivals.

The need to use memory is, by now, well-justified. One source of randomness available in switches is that which is in the arrivals process. Using arrivals to find matchings also has the big benefit of providing information about recently loaded, and hence likely heavy VOQs. (At least these VOQs will certainly be nonempty!)

Since the edges which receive an arrival at a given time will not necessarily form a matching, the MERGE procedure we have used in LAURA will not be directly usable for SERENA. A simple modification of the MERGE procedure leads to the ARR-MERGE procedure described below.

1) The Serena Algorithm: Let $S(t)$ be the matching used by SERENA at time t. Let $A(t+1) = [A_{ij}(t+1)]$ denote the arrival graph, where $A_{ij}(t+1) = 1$ indicates arrival at VOQ$_{ij}$. At time $t+1$:

a) compute $S(t+1) = $ Arr-Merge$(S(t), A(t+1))$;
b) use $S(t+1)$ as the schedule.

The Arr-Merge Procedure (Fig. 10): Let M denote the schedule used at time t, and let A denote the subgraph induced by packets arriving at time $t+1$. Let $G = M \cup A$ be the subgraph induced by the edges of M and A on the bipartite graph consisting of input and output nodes. As in the MERGE procedure of LAURA, the goal of ARR-MERGE is to find a maximum weight matching $\tilde{M}$, on G. Whereas, M is a matching, A is not necessarily a matching. This is because multiple edges can be incident on the same output node due to multiple arrivals to that output. Therefore, we cannot simply combine M and A using

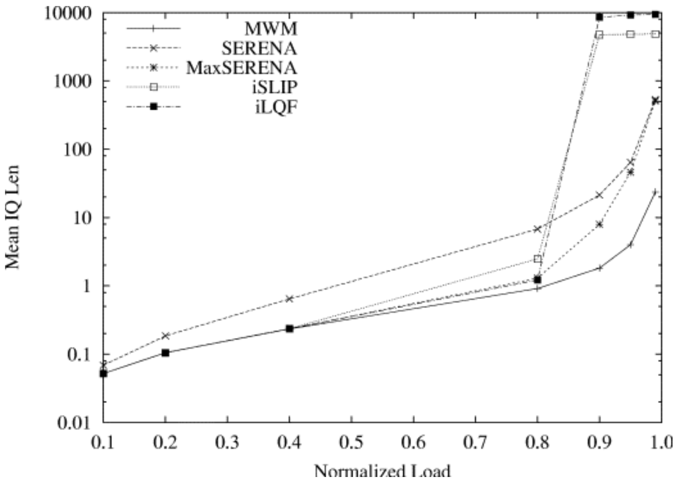

Fig. 11. Mean IQ length under diagonal traffic.

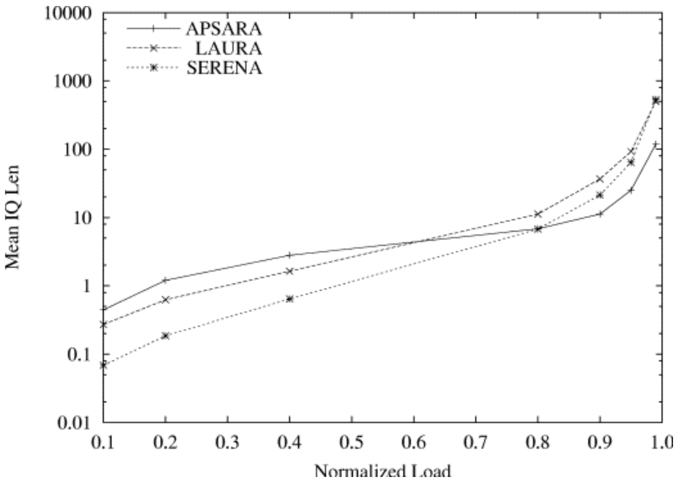

Fig. 12. Mean IQ length under diagonal traffic.

the MERGE procedure. We need to consider the following two cases.

Case 1) A is a matching. This is a simple case, Arr-Merge reduces to MERGE on (M, A), yielding the matching $\tilde{M}$.

Case 2) A is not a matching. Let $\mathcal{U}^\star$ denote collection of outputs which have one or more arrival edges incident on them. For every $u \in \mathcal{U}^\star$ do the following: among the arrival edges incident on output u, pick the edge with the highest weight and discard the remaining edges. At the end of this process, each output in $\mathcal{U}^\star$ is matched with exactly one input.

To complete the matching A, connect the remaining input–output pairs by adding edges in a round-robin fashion, without considering their weights. The round-robin mechanism avoids queue starvation and provides fairness among queues which are not receiving arrival. Call the resulting complete matching $\tilde{A}$. Now ARR-MERGE reduces to MERGE on $(M, \tilde{A})$, yielding matching $\tilde{M}$.

Theorem 7: SERENA is stable under all admissible Bernoulli i.i.d. inputs.

Proof: Again, this follows from Theorem 2, since the probability that the arrival graph at any time t will be equal to the maximum weight matching is lower bounded by some constant $c > 0$. This is sufficient to establish the stability of SERENA. □

Performance: The simulation setting is identical to that of the APSARA algorithm. SERENA is compared with the MWM, iSLIP, and iLQF algorithms under diagonal traffic. The results are shown in Fig. 11. The algorithms SERENA and MAXSERENA (the maximized version of SERENA) perform quite competitively with respect to MWM.

Finally, Fig. 12 compares the three algorithms we have proposed—APSARA, LAURA and SERENA—under diagonal traffic. All these algorithms perform competitively with each other, showing very good delays. SERENA, which uses randomness from arrivals, performs better than LAURA for all loads, showing the usefulness of using information from arrivals. For lower loads, APSARA performs the worst but for higher loads, it outperforms both SERENA and LAURA.

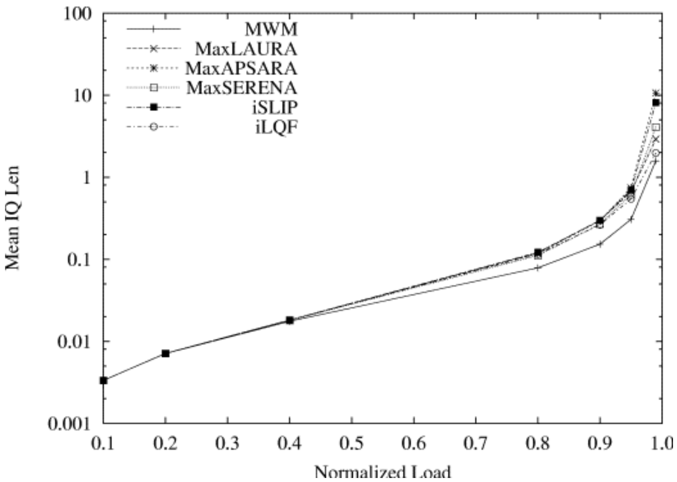

Fig. 13. Mean IQ length for uniform traffic.

Fig. 13 shows that all the algorithms considered are well-behaved under uniform traffic.

SERENA: Complexity: All of the work done by SERENA is in the ARR-MERGE procedure. It is not hard to see that the complexity of ARR-MERGE is $O(N)$. Indeed, ARR-MERGE only needs to perform the following simple operations: 1) break ties at outputs for which there is more than one arrival; 2) maximize the resulting arrival graph (indiscriminately, if need be); and 3) MERGE. Since all of these operations are simple to implement, and the performance of SERENA, we prefer SERENA to LAURA.

IV. GENERAL STABILITY CONDITIONS

The scheduling algorithms discussed in this paper are proved to be stable under Bernoulli i.i.d. arrival traffic. It is not clear apriori how the algorithms APSARA, SERENA, or LAURA would behave under admissible correlated traffic. This leads us to study the rate stability of these algorithms using fluid-model developed in [5]. We first present basic notations and definitions from [5].

Recall that $A(t) = [A_{ij}(t)]$ and $Q(t) = [Q_{ij}(t)]$ denote the discrete-time arrival process and queue sizes as defined before. Let the cumulative arrival process be denoted as

$$\bar{A}(t) = \left[\bar{A}_{ij}(t)\right] \triangleq \left[\sum_{s=0}^{t} A_{ij}(s)\right].$$

Let $D(t) = [D_{ij}(t)]$ denote the cumulative departure process, that is, total departures occurred till time t. Consider the following definitions.

Definition 3: A cumulative arrival process $\bar{A}(t)$ is called *rate admissible* if the following conditions are satisfied.

1) $\bar{A}(t)$ satisfies the strong law of large numbers; and let $\lim_{t\to\infty} \bar{A}_{ij}(t)/t = \lambda_{ij} \quad \forall i, j \quad$ w.p.1;
2) $\sum_i \lambda_{ij} \leq 1$ and $\sum_j \lambda_{ij} \leq 1$.

Definition 4: A switch is said *rate stable* if, with probability one

$$\lim_{t\to\infty} \frac{D_{ij}(t)}{t} = \lambda_{ij}, \quad \forall i, j$$

for any rate admissible arrival process.

Consider the following rate-stable version of Lemma 1 which is proved in [8]:

Lemma 2: Consider an input-queued switch with arbitrary arrival traffic. Let $Q(t)$ be the queue-size process that results when the switch uses scheduling algorithm B. Let $W^B(t)$ denote the weight of the schedule used by B at time t, and let $W^\star(t)$ be the weight of MWM given the same queue-size process $Q(t)$. Let $T(t) = W^\star(t) - W^B(t)$. If there exists a positive constant c such that

$$E\left[T(t)\right] < c \quad \forall t \tag{6}$$

then, the algorithm B is rate stable if the arrival traffic is rate admissible.

We would like to apply the result of Lemma 2 to obtain rate stability of the proposed algorithms. The property (6) holds for APSARA and LAURA because of Hamiltonian walk and random sampling, respectively, and hence, it is independent of the type of arrival traffic. For exactly the same reason as for LAURA, the property (6) holds for algorithm ALGO2 proposed by Tassiulas [27] too. Thus, APSARA, LAURA, and ALGO2 are rate stable. For SERENA, property (6) is true only if the arrival process is stationary and independent between inputs. It does not require independence of the arrival processes in time. Thus, we obtain the following theorem:

Theorem 8: Under any rate-admissible traffic APSARA, LAURA and ALGO2 are rate stable. Further, if the traffic is such that the arrival process is stationary and independent between inputs, then SERENA is also rate stable.

Proof: To prove the stability for these algorithms, we would like to use Lemma 2. To do this, we need to prove that for all these algorithms, property (6) is true. We check this as follows.

1) APSARA: The proof of Theorem 3 shows that the APSARA (with Hamiltonian walk) schedule has weight at most $2NN!$ smaller than the weight of MWM schedule. This shows that property (6) holds.
2) LAURA and ALGO2: This can be proved similarly to the argument for APSARA in Theorem 3. First note that both algorithms LAURA and ALGO2 have the random sampling procedure which will guarantee that at any time the probability of schedule to be MWM will be at least $\delta > 0$. Thus, on average every $1/\delta$ times the schedule becomes of same weight as MWM. Between two consecutive time slots, the difference between the weight of MWM schedule and the schedule obtained by LAURA (or ALGO2) can at most increase by $2N$. Thus, on average at any time t, the weight difference between MWM schedule and the schedule obtained by LAURA (or ALGO2) is bounded above $2N/\delta$. This proves the desired property (6).
3) SERENA: as in case 1), if any algorithm has positive probability of having MWM as a schedule every time, the algorithm has property (6). Under SERENA, the new schedule is obtained by the arrival occurred in previous time. If the traffic is independent between inputs and stationary, then the probability of any matching formed using arrivals is positive bounded away from zero. This guarantees the desired property (6).

From above and Lemma 2, we obtain the rate stability of APSARA, LAURA, and ALGO2 under any admissible traffic. Under additional conditions on arrival traffic of stationarity and independence between inputs, SERENA is rate stable. □

A. Simulation Study under Correlated Traffic

The algorithms discussed in this paper all try to learn the weight of the MWM schedule. Hence, intuitively, temporal correlation in traffic could help these algorithms to learn quicker and achieve better performance relative to MWM. Stability for correlated traffic is guaranteed by Theorem 8, and we study the effect of correlation on delays by simulations.

We consider the same simulation setting as in Section III-A5, but now the traffic is generated according to correlated "bursty" traffic. The cell arrival process at each input i is characterized by a two-state ON–OFF model.

- **ON state**. As soon as the input has just entered this state, it chooses one random destination weighted by the elements of the row corresponding to that input in the traffic matrix. When input i is in this state for the current time slot, a packet is generated. All packets, generated during a single visit of input i to the ON stage, have the same destination. The duration in time slots of the ON state is geometrically distributed with a given mean.
- **OFF state**. In this state, no cells are generated. The probability that the OFF state lasts j slots is
$$P(j) = p(1-p)^j, \qquad j \geq 0.$$
The average idle period duration in slots is $\mathrm{E}_{\mathrm{OFF}} = (1-p)/p$. The parameter p is set so as to achieve the desired input load.

In other words, the above model generates packets destined for the same output with a geometric burst size. The average burst size is an indicator of the temporal correlation in the traffic. Fig. 14 shows the mean IQ length for the proposed scheduling algorithms as a function of the average burst size. The input load

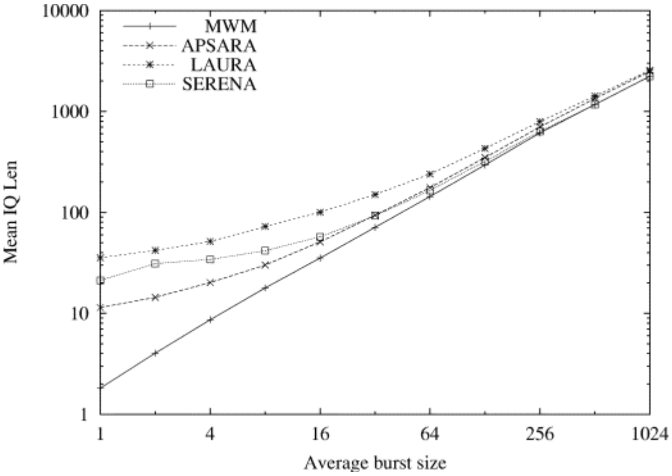

Fig. 14. Mean IQ length under diagonal traffic, when the traffic is correlated according to a bursty ON/OFF source. The input load is 0.9.

is set equal to 0.9. Note that the case with average burst size equal to 1 correspond to the result for i.i.d. Bernoulli traffic, shown in Fig. 12, for normalized load 0.9.

All the three proposed algorithms behave closer to the MWM as the average burst size (i.e., the degree of correlation in the traffic) increases. Correlation can indeed help, since the correlation among subsequent maximum weight matchings is captured by the memory retained in the previous matching.

Particular attention should be paid to SERENA, whose performance could degrade if correlation among different inputs is allowed (note that Theorem 8 is not guaranteed to hold in this case). For example: at time t all the inputs receive packets destined for the $(t \bmod N)$th output. The arrival graph, after the ARR-MERGE procedure, will degenerate in only one edge, corresponding to one single arrival and this fact can considerably degrade the performance of SERENA. In this paper, we do not consider the effect of correlation among different inputs, since it is not a realistic scenario in a large network.

V. CONCLUSION

The paper presented some new approaches for designing simple, high-performance schedulers for high-aggregate bandwidth switches. The following general features of the switch scheduling problem were exploited: 1) the use of memory; 2) the randomized weight augmentation; and 3) the randomness and the information provided by recent arrivals.

We have presented a derandomized algorithm and established its stability using methods which may apply more widely. Three algorithms—APSARA, LAURA, and SERENA—were developed to exploit the above-mentioned features. These algorithms are stable under any admissible arrival process and are robust to correlated traffic. Simulations show that they outperform some other known algorithms in terms of delay, and perform competitively with respect to the maximum weight matching algorithm. While the algorithms proposed exploit different features, implementations could easily combine these features.

REFERENCES

[1] M. M.Marsan M. Ajmone, A. Bianco, E. Leonardi, and L. Milia, “RPA: a flexible scheduling algorithm for input buffered switches,” *IEEE Trans. Commun.*, vol. 47, pp. 1921–1933, Dec. 1999.

[2] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, “High speed switch scheduling for local area networks,” *ACM Trans. Comput. Syst.*, vol. 11, pp. 319–351, Nov. 1993.

[3] C. S. Chang, D. S. Lee, and Y. S. Jou, “Load balanced Birkhoff–von Neumann switches,” in *Proc. 2001 IEEE Workshop on High Performance Switching and Routing*, 2001, pp. 276–280.

[4] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, “Matching output queueing with a combined input output queued switch,” *IEEE J. Select. Areas Commun.*, vol. 17, pp. 1030–1039, 1999.

[5] J. Dai and B. Prabhakar, “The throughput of data switches with and without speedup,” in *Proc. IEEE INFOCOM 2000*, vol. 2, Tel-Aviv, Israel, Mar. 2000, pp. 556–564.

[6] H. Duan, J. W. Lockwood, S. M. Kang, and J. D. Will, “A high performance OC12/OC48 queue design prototype for input buffered ATM switches,” in *Proc. IEEE INFOCOM’97*, vol. 1, Kobe, Japan, 1997, pp. 20–28.

[7] R. Durrett, *Probability: Theory and Examples*, 2nd ed. Belmont, CA: Duxbury, 1995.

[8] Y. Ganjali, A. Kesavarzian, and D. Shah, “Input queued switch: packet v/s cell scheduling,” presented at the IEEE INFOCOM’03, San Francisco, CA, Apr. 2003.

[9] P. Giaccone, D. Shah, and B. Prabhakar, “An implementable parallel scheduler for input-queued switches,” in *Proc. Hot Interconnects 9*, Stanford, CA, Aug. 2001, pp. 9–14.

[10] P. Giaccone, “Queueing and scheduling algorithms for high performance routers,” Ph.D. dissertation, Politecnico di Torino, Italy, Feb. 2002.

[11] M. W. Goudreau, S. G. Kolliopoulos, and S. B. Rao, “Scheduling algorithms for input-queued switches: randomized techniques and experimental evaluation,” in *Proc. IEEE INFOCOM 2000*, vol. 3, Tel-Aviv, Israel, Mar. 2000, pp. 1634–1643.

[12] S. Iyer and N. McKeown, “Making parallel packet switches practical,” in *Proc. IEEE INFOCOM’01*, vol. 3, Anchorage, AK, Apr. 22–26, 2001, pp. 1680–1687.

[13] S. Keshav and R. Sharma, “Issues and trends in router design,” *IEEE Commun. Mag.*, vol. 36, pp. 144–151, May 1998.

[14] P. Krishna, N. S. Patel, A. Charny, and R. J. Simcoe, “On the speedup required for work-conserving crossbar switches,” *IEEE J. Select. Areas Commun.*, vol. 17, pp. 1057–1066, June 1999.

[15] E. Leonardi, M. Mellia, F. Neri, and M. A. Marsan, “Bounds on average delays and queue size averages and variances in input queued cell-based switches,” in *Proc. IEEE INFOCOM 2001*, vol. 3, Anchorage, AK, Apr. 22–26, 2001, pp. 1095–1103.

[16] N. McKeown, V. Anantharan, and J. Walrand, “Achieving 100% throughput in an input-queued switch,” in *Proc. IEEE INFOCOM’96*, vol. 1, San Francisco, CA, Mar. 1996, pp. 296–302.

[17] N. McKeown, “Scheduling algorithms for input-queued cell switches,” Ph.D. dissertation, Univ. California, Berkeley, CA, 1995.

[18] ——, “iSLIP: a scheduling algorithm for input-queued switches,” *IEEE Trans. Networking*, vol. 7, pp. 188–201, Apr. 1999.

[19] A. Mekkittikul and N. McKeown, “A practical scheduling algorithm to achieve 100% throughput in input-queued switches,” in *Proc. IEEE INFOCOM ’98*, vol. 2, Apr. 1998, pp. 792–799.

[20] M. Mitzenmacher, “The power of two choices in randomized load balancing,” Ph.D. dissertation, Univ. California, Berkeley, CA, 1996.

[21] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1995.

[22] A. Nijenhuis and H. Wilf, *Combinatorial Algorithms: for Computers and Calculators*, 2nd ed. New York: Academic, 1978, ch. 7, p. 56.

[23] K. Psounis and B. Prabhakar, “A randomized web-cache replacement scheme,” in *Proc. IEEE INFOCOM’01*, Anchorage, AK, Apr. 22–26, 2001, pp. 1407–1415.

[24] B. Prabhakar and N. McKeown, “On the speedup required for combined input and output queued switching,” *Automatica*, vol. 35, no. 12, pp. 1909–1920, 1999.

[25] D. Shah, P. Giaccone, and B. Prabhakar, “An efficient randomized algorithm for input-queued switch scheduling,” in *Proc. Hot Interconnects 9*, Stanford, CA, Aug. 2001, pp. 3–8.

[26] Y. Tamir and H.-C. Chi, “Symmetric crossbar arbiters for VLSI communication switches,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 13–27, Jan. 1993.

[27] L. Tassiulas, "Linear complexity algorithms for maximum throughput in radio networks and input queued switches," in *Proc. IEEE INFOCOM'98*, vol. 2. New York, 1998, pp. 533–539.

[28] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Trans. Automat. Contr.*, vol. 37, pp. 1936–1948, Dec. 1992.

Paolo Giaccone (S'99–M'02) received the Dr.Ing. and Ph.D. degrees in telecommunications engineering from the Politecnico di Torino, Torino, Italy, in 1998 and 2001, respectively.

He is an Assistant Professor in the Electronics Department, Politecnico di Torino. During the summer of 1998, he visited the High-Speed Networks Research Group, Lucent Technologies, Holmdel, NJ. During 2000–2001 and during the summer of 2002, he visited Prof. B. Prabhakar in the Electrical Engineering Department, Stanford University, Stanford, CA. Between 2001–2002, he held a Postdoctoral position at the Politecnico di Torino and during the summer of 2002 at Stanford University. His main area of interest is the design of scheduling policies for high-performance routers.

Balaji Prabhakar (M'00) is an Assistant Professor of Electrical Engineering and Computer Science, Stanford University, Stanford, CA. He is interested in network algorithms (especially for switching, routing, and bandwidth partitioning), wireless networks, web caching, network pricing, information theory, and stochastic network theory.

Dr. Balaji has received the CAREER Award from the National Science Foundation, the Erlang Prize from the INFORMS Applied Probability Society, and the Rollo Davidson Prize from the University of Cambridge. He has been a Terman Fellow at Stanford University and a Fellow of the Alfred P. Sloan Foundation.

Devavrat Shah received the B.Tech. degree from the Indian Institute of Technology (IIT), Bombay, India, in 1999. He is working toward the Ph.D. degree in the Computer Science Department, Stanford University, Stanford, CA.

He is interested in design and analysis of network algorithms, analysis of stochastic networks, and information theory.

Mr. Devavrat received the President of India Gold Medal from IIT in 1999.