Towards Simple, High-performance Schedulers for High-aggregate Bandwidth Switches

Paolo Giaccone, Balaji Prabhakar, Devavrat Shah
giaccone@polito.it, balaji@isl.stanford.edu, devavrat@cs.stanford.edu
Dept. of EE, Politecnico di Torino; Depts. of EE and CS, Stanford University; Dept. of CS, Stanford University

Abstract— **High-aggregate bandwidth switches are those whose port count multiplied by the operating line rate is very high; for example, a 30 port switch operating at 40 Gbps or a 1000 port switch operating at 1 Gbps. Designing high-performance schedulers for such switches is a challenging problem for the following reasons: (i) High performance requires finding good matchings, (ii) good matchings take time to find, and (iii) 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: (a) 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, (b) 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 competitive with respect to the maximum weight matching. The main contribution of this paper is a suite of simple to implement, high-performance scheduling algorithms for IQ switches. The stability proof involves a derandomization procedure and uses methods which may have wider applicability.

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 to that of an output-queued or a shared-memory architecture.

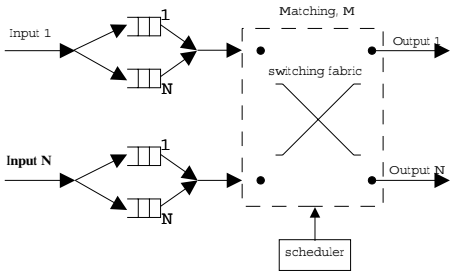

Fig. 1. Logical structure of an input-queued cell switch

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 virtual output queues 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.

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 algorithm (MWM). Given a weighted bipartite graph, the MWM finds that matching whose weight is the highest. For example, Figure 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.

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/sec is defined to be the product NL bits/sec. 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 [10], 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

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

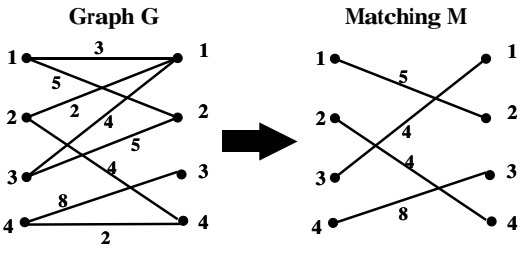

Fig. 2. Example of weighted bipartite graph and its maximum weight matching.

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 [13], [25], showed that under Bernoulli i.i.d. packet arrival processes the MWM is stable so long as no input or output is oversubscribed [2]. More recently, other algorithms have been proposed for providing exact delay bounds [4], [11], [21]. Those algorithms in fact provide something much stronger: they allow a switch whose fabric runs at a speedup of between 2 and 4 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], [11], [21] 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 [15], iLQF [14], RPA [1], MUCS [6] and WFA [23]. However, these algorithms perform poorly compared to MWM when the input traffic is non-uniform: 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 [3], [9] and proven to be stable. But, [3] introduces an extra packet resequencing problem and [9] 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 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.

[2]The weights were taken to be the length of Q_{ij} originally and later work [16] took the weights to be the age of the oldest packet in Q_{ij}.

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.
a. 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.
b. 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 [18] for a general exposition of randomized algorithms, [24], [8] for application to switching, and [17], [20] for other applications to networking.

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 that algorithms based only upon random samples are unstable, making it necessary to use memory. We recall the recent work of Tassiulas [24], 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.

As a final comment, recall that high aggregate bandwidth switches come in two flavors: core and edge. In Section III we shall comment upon the suitability of the algorithms we propose for use in either of the two types of switch. We shall also present variants of the basic algorithms, to better suit the type of switch being designed.

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

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

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 [24].

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 [24]). 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 $t = 1, \ldots, N!$. We extend $Z(t)$ for $t > N!$ by defining $Z(t) = Z(t \mod N!)$. One simple algorithm for such a Hamiltonian walk is described, for example, in Chapter 7 of [19]. 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: $Z(1) = (1, 2, 3)$, $Z(2) = (1, 3, 2)$, $Z(3) = (3, 1, 2)$, $Z(4) = (3, 2, 1)$, $Z(5) = (2, 3, 1)$, $Z(6) = (2, 1, 3)$, $Z(7) = Z(1)$, and $Z(8) = Z(2), ...$

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 [12], [13]) for some $\delta > 0$ and $K > 0$

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

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

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

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

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

Hence, we obtain

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

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

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

Since the arrival rate matrix, Λ, is admissible it is strictly doubly sub-stochastic. Therefore, from arguments made in Lemma 2 of [13], 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

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

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

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

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 1 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 3 neighbors S_1, S_2 and S_3 are given below:

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

A.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:
(i) Determine $\mathcal{N}(S(t))$ and $Z(t+1)$.
(ii) 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 $S' \in \mathcal{M}(t+1)$.
(iii) 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 neighbourhood set. In particular, we are aiming for a neighbourhood 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 $i_2 \stackrel{\triangle}{=} (i_1 + 1 \mod N)$, 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 $\mathcal{N}_L(S)$.

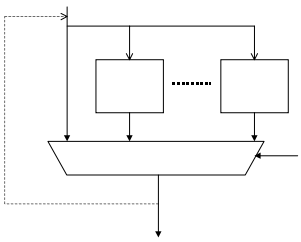

Fig. 3. A schematic for the implementation of APSARA. The old matching, $S(t)$, and the new arrivals, $A(t+1)$, are used to compute the weights of the k neighbor matchings in parallel. The new matching, $S(t+1)$, is the one with highest weight among all the neighbors. Note that this architecture is parallel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

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 the 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.

A.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:
(i) 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.
(ii) 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 $S' \in \mathcal{M}_K(t+1)$.
(iii) $S(t+1) = \arg \max_{S' \in \mathcal{M}_K(t+1)} \{W_{S'}(t+1))\}$.

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 0. 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.

A.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, $W^S(t) \geq \frac{1}{2}W^\star(t)$ and the theorem is proved. □

A.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].

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

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

A.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. Figure 10 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 [7], [22].

Switch: No. 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 $|k| = (k \mod N)$. 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 figure 10).

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 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.

Figure 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 10 packets on average.

We see that APSARA-L only 32 modules, performs quite well when compared to APSARA, which uses $\binom{32}{2} = 496$ modules; even at high loads, the difference between queue sizes is

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

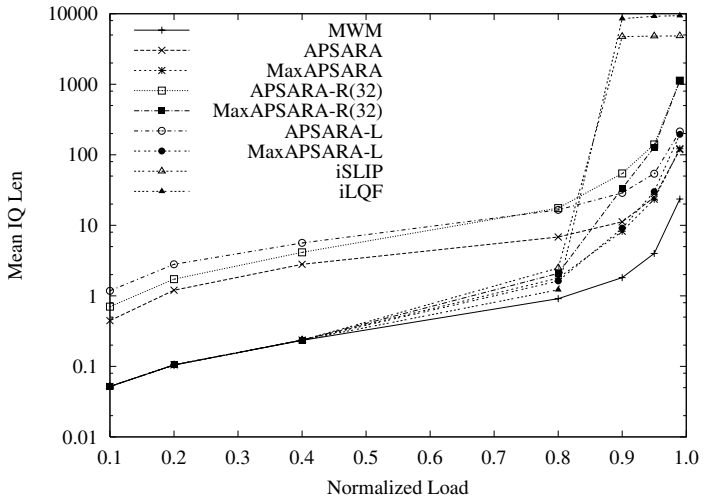

Fig. 4. Mean IQ length for APSARA under diagonal traffic.

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 [24], ALGO2 provides 100% throughput. However, its delay performance is quite poor (as we will see in Figure 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 (non-uniformly) randomly chosen matching.

There are three main features in the design of LAURA.

1. Use of memory.

2. Non-uniform random sampling.

3. A merging procedure for weight augmentation.

B.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 $t+1$.

The RANDOM Procedure

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

RANDOM 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 $\log_2 N$:

(i) Let i be the current iteration number. Let $k \le N$ 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.

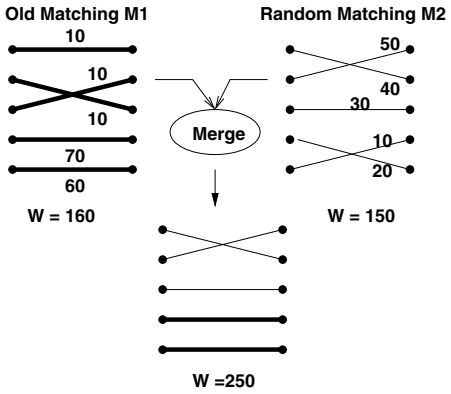

Fig. 5. An illustration of the MERGE applied to matchings M1 and M2. The final matching is the maximum weight matching on the subgraph defined by edges of M1 and M2.

(ii) 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 $S_i(k)$.

The 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 covers 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, $C_1, ..., C_m$ 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

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

Figure 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 $M2$.

B.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. □

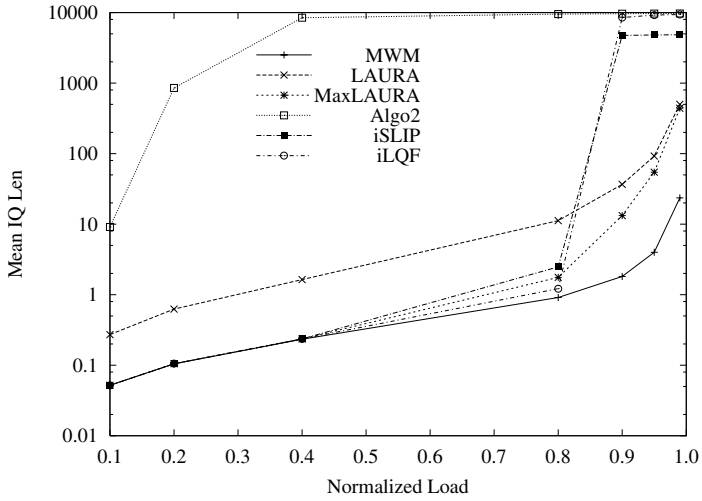

Fig. 6. Mean IQ length for LAURA under diagonal traffic.

B.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 Figure 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 lows and comparable with the MWM algorithm.

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.

C.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 $t+1$,

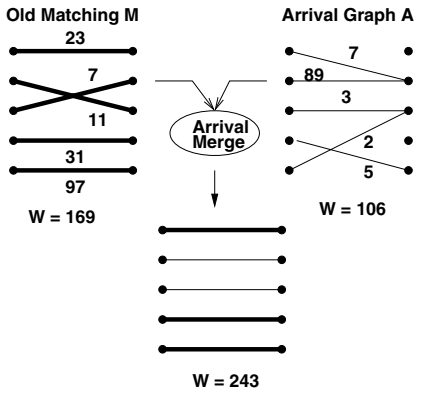

Fig. 7. An illustration of the ARR-MERGE procedure, given the matching M and the arrival graph A.

(a) Compute $S(t+1) = \text{ARR-MERGE}(S(t), A(t+1))$.
(b) Use $S(t+1)$ as the schedule.

The ARR-MERGE Procedure

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 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. □

C.2 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

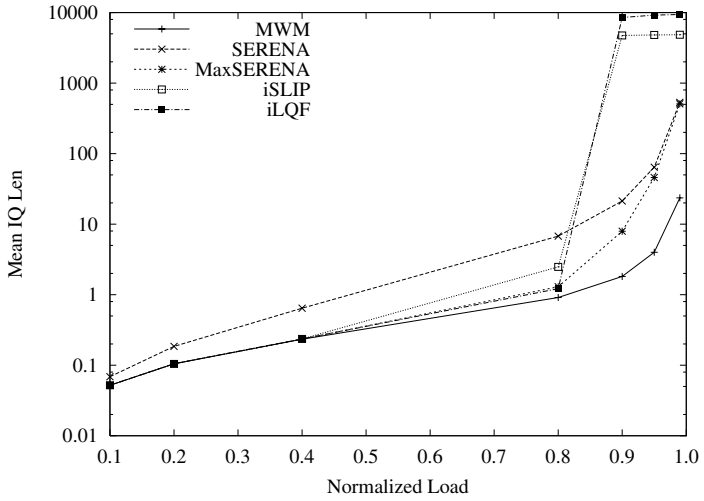

Fig. 8. Mean IQ length under diagonal traffic.

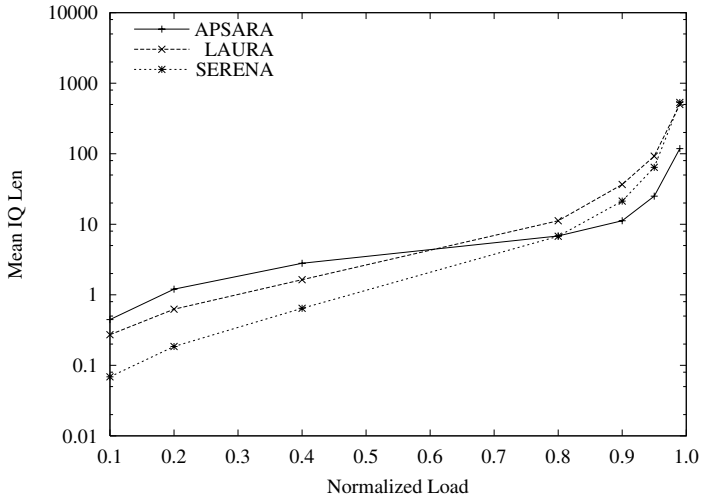

Fig. 9. Mean IQ length under diagonal traffic.

in Figure 8. The algorithms SERENA and MaxSERENA (the maximized version of SERENA) perform quite competitively with respect to MWM.

Finally, Figure 9 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.

Figure 10 shows that all the algorithms considered are well-behaved under uniform traffic.

C.3 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: (a) Break ties at outputs for which there is more than one arrival, (b) maximize the resulting arrival graph (indiscriminately, if need be), and (c) MERGE. Since all of these operations are simple to implement, and the performance of SERENA we prefer SERENA to LAURA.

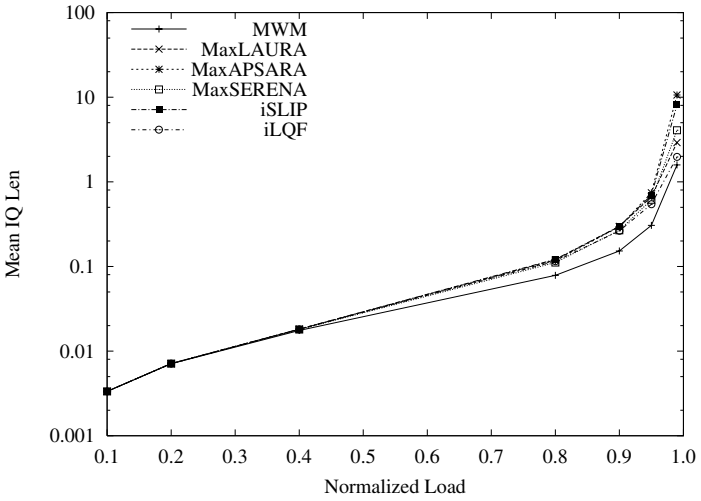

Fig. 10. Mean IQ length for uniform traffic.

IV. CONCLUSIONS

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: (i) The use of memory, (ii) the randomized weight augmentation, and (iii) 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 admissible, Bernoulli i.i.d. inputs. Simulations show that they outperform some other known algorithms in terms of delay, and perform competitively with respect to the maximum weight matching algorithm.

REFERENCES

[1] Ajmone Marsan M., Bianco A., Leonardi E., Milia L., "RPA: a flexible scheduling algorithm for input buffered switches", *IEEE Trans. on Communications*, vol. 47, n. 12, Dec. 1999, pp. 1921-33

[2] Anderson T., Owicki S., Saxe J., Thacker C., "High speed switch scheduling for local area networks", *ACM Trans. Comput. Syst.*, vol 11, n. 4, Nov.1993, pp. 319-351

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

[4] Chuang S.-T., Goel A., McKeown N., Prabhakar B., "Matching output queueing with a combined input output queued switch", *IEEE Journal on Selected Areas in Communications*, vol. 17, n. 6, 1999, pp. 1030-1039

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

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

[7] Giaccone P., Shah D., Prabhakar B., "An implementable parallel scheduler for input-queued switches", Hot Interconnects 9, Stanford, CA, Aug. 2001, pp. 9-14

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

[9] Iyer S., McKeown N., "Making Parallel Packet Switches Practical", *IEEE INFOCOM'01*, Alaska, USA, Mar. 2001

[10] Keshav S., Sharma R., "Issues and trends in router design", *IEEE Communications Magazine*, vol. 36, n. 5, May 1998, pp.144-151

[11] Krishna P., Patel N.S., Charny A., Simcoe R.J., " On the speedup required

for work-conserving crossbar switches", *IEEE Journal on Selected Areas in Communications*, vol. 17, n. 6, June 1999, pp. 1057-1066

[12] Leonardi E., Mellia M., Neri F., Marsan M. A., "Bounds on Average Delays and Queue Size Averages and Variances in Input Queued Cell-Based Switches", *IEEE Infocom 2001*, Alaska, pp. 1095 -1103 vol.3, April 22 - 26, 2001.

[13] McKeown N., Anantharan V., Walrand J., "Achieving 100% throughput in an input-queued switch" *IEEE INFOCOM'96*, vol. 1, San Francisco, CA, Mar. 1996, pp. 296-302

[14] McKeown N., "Scheduling algorithms for input-queued cell switches", *Ph.D. Thesis*, University of California at Berkeley, 1995

[15] McKeown N., "iSLIP: a scheduling algorithm for input-queued switches", *IEEE Trans. on Networking*, vol. 7, n. 2, Apr. 1999, pp. 188-201

[16] Mekkittikul A., McKeown N., "A practical scheduling algorithm to achieve 100% throughput in input-queued switches",*IEEE INFOCOM'98*, vol. 2, , April. 1998, pp. 792-799

[17] Mitzenmacher M., "The power of two choices in randomized load balancing", *Ph.D. thesis*, University of California at Berkeley, 1996

[18] Motwani R., Raghavan P., "Randomized algorithms", *Cambridge Univ. Press*, 1995

[19] Nijenhuis A., Wilf H., "Combinatorial algorithms: for computers and calculators", 2^{nd} *Edition, Academic Press*, chap. 7, New York, 1978, p. 56

[20] Psounis K., Prabhakar B., "A randomized web-cache replacement scheme", *IEEE INFOCOM'01*, Anchorage, AK, April 22-26, 2001

[21] Prabhakar B., McKeown N., "On the speedup required for combined input and output queued switching", *Automatica*, vol 35, n. 12, 1999, pp. 1909-1920

[22] Shah D., Giaccone P., Prabhakar B., "An efficient randomized algorithm for input-queued switch scheduling", Hot Interconnects 9, Stanford, CA, Aug. 2001, pp. 3-8

[23] Tamir Y., Chi H.-C., "Symmetric crossbar arbiters for VLSI communication switches", *IEEE Transaction on Parallel and Distributed Systems*, vol. 4, n. 1, Jan. 1993, pp. 13-27

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

[25] Tassiulas L., Ephremides. A., "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks". *IEEE Trans. on Automatic Control*, vol. 37, n. 12, Dec. 1992, pp. 1936-1948